PARALLEL ADA IMPLEMENTATION OF A MULTIPLE MODEL KALMAN FILTER TRACKING SYSTEM: A SOFTWARE ENGINEERING APPROACH

THESIS

Walter John Lemanski
Captain, USAF

AFIT/GCS/ENG/89M-2

DEPARTMENT OF THE AIR FORCE

AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/89M-2

PARALLEL ADA IMPLEMENTATION OF A MULTIPLE MODEL KALMAN FILTER TRACKING SYSTEM: A SOFTWARE ENGINEERING APPROACH

THESIS

Walter John Lemanski
Captain, USAF

AFIT/GCS/ENG/89M-2

Approved for public release; distribution unlimited

AFIT/GCS/ENG/89M-2

PARALLEL ADA IMPLEMENTATION OF A MULTIPLE MODEL KALMAN FILTER TRACKING SYSTEM: A SOFTWARE ENGINEERING APPROACH

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Engineering

Walter John Lemanski, B.S.

Captain, USAF

March, 1989

Acknowledgments

Much credit for this effort belongs to my thesis committee, especially my chairman, Dr Thomas Hartrum. In addition to his technical help, he willingly suffered through literally hundreds of pages of drafts and revisions, tens of hours of complaints and misgivings, and dozens of passed deadlines. Through it all, he was always there to give support when the outlook was bleak. Most of all, he did what a thesis advisor should do: he let me do it my way. Thanks also goes to my other committee members: Dr Peter Maybeck and Major Dave Umphress. Dr Maybeck's success in imbuing this control systems neophyte with sufficient knowledge to complete this type of research is a true credit to his teaching ability. Major Umphress' willingness to spend hours helping me find unorthodox errors in even more unorthodox code is a tribute not only to his knowledge of the Ada language, but his patience as well. I especially appreciate the continued patience and understanding received from all three of them each of the many time the "light at the end of the tunnel" turned out to be a train.

Special thanks also goes to Captain John Norton, AFIT class of 1988, for the tremendous amount of time and effort he spent with me explaining the intricacies of the Kalman filter algorithm and the FORTRAN program. Despite the fact that he was trying to complete his own thesis, he was always willing to stop whatever he was doing to help me with a problem. I would have never completed this project without his help.

With that, another milestone on the highway of life has been passed with the hope that the road not yet traveled holds rewards of a different kind.

It's a Grand Illusion

Walter John Lemanski

Table of Contents

List of Figures

Abstract

The success of the Strategic Defense Initiative depends directly on significant advances in both computer hardware and software development technologies. Parallel architectures and the Ada programming language have advantages that make them candidates for use in SDI command and control computer systems. This thesis examines those advantages in the context of an SDI-type application: the implementation of a Kalman filter tracking system.

This research consists of three parts. The first is a set of software engineering guidelines developed for use in creating parallel designs suitable for implementation in Ada. These guidelines cover the design process from initial problem analysis to final detailed design. Methods of problem decomposition are discussed, as are language partitioning strategies. Justification is provided for using the Ada task construct for process boundaries, and Ada multitasking design issues are reviewed. A parallel software design methodology is also described.

The second part is an illustration of the use of these software engineering guidelines to design a multiple model adaptive Kalman filter based tracking system and its associated simulation program. Theoretical research, currently being done at the Air Force Institute of Technology, is used as a basis for this phase. The result of each step of the guidelines is documented.

The final part is a parallel implementation of the Kalman filter tracking system design, including several reusable matrix operation routines, on the Encore Multimax 320 in Ada. Implementation and testing techniques are documented, as are problems that were encountered with the design and the Encore Concurrent Ada development environment. Timing results are analyzed to determine the efficiency of the design and implementation, as well as the suitability of the particular tracking algorithm for parallelization. Additional analysis is included on the adequacy of the software engineering guidelines developed in part one and on the results of using Ada in a true parallel environment.

PARALLEL ADA IMPLEMENTATION OF A MULTIPLE MODEL KALMAN FILTER TRACKING SYSTEM: A SOFTWARE ENGINEERING APPROACH

I. Introduction

The Strategic Defense Initiative, inaugurated by President Reagan to develop a defensive shield against ballistic missile attack, represents one of the most monumental research efforts ever undertaken by this country. Its complexity and the demands it places on the technological state of the art rival, if not exceed, those of the Manhattan Project and the Apollo moon landings. Successful realization of such a defense will require significant advances in many areas of science and engineering, but it is now generally accepted that computers and their associated software have become the "long pole in the tent."

1.1 Software and the Strategic Defense Initiative

The feasibility of the Strategic Defense Initiative is directly dependent on the successful implementation of its command and control computer systems. Human reaction times are simply too slow to manage a battle in space. The SDI defense system will depend on computers to detect missile firings, compute attack trajectories, descriminate between warheads and decoys, and aim and fire its weapons (Waldrop, 1986:710). Without computers, no part of the system will function, and without correct software, none of the computers will function. The Eastport Study Group, convened to review the role of software in the total SDI effort, made this clear in their finding that "the anticipated complexity of the battle management software and the necessity to test, simulate, modify, and evolve the system make battle management and command, control, and communication (BM/C^3) the paramount strategic defense problem" (Eastport Study Group, 1985:v).

The reason for this emphasis on software is clear. "The Star Wars battle management system will be by far the most complex body of software ever devised. By one estimate

it will require up to 100 million lines of computer code written by hundreds of thousands of individual programmers" (Waldrop, 1986:710). Other estimates are not as large, but they show the same trends. "Estimates of its size range from 10 million to 30 million lines of executable source code. Most other systems that work in a distributed real-time environment like SDI are very much smaller – under a million lines of code" (Ramamoorthy, 1986:64).

There are factors other than sheer size that also add to the expected complexity. The system will have to run in real time, and there will be absolute deadlines for processing requirements (Parnas, 1985:1328). Real-time systems have important differences from more standard types of computing. "A real-time computing system accepts data from an external activity, analyzes the data in real time, and provides output that influences an external process" (McGee, 1987:4). These two requirements, interaction with devices external to the computing system and critical time constraints, cause major development difficulties that do not exist in non-real-time software systems. Time constraints must be met; otherwise vital data may be lost or critical control actions may not be taken in time to have the desired effect. Interaction with external devices is complicated by their different speeds and control requirements. Concurrent control is necessary to ensure that the speed of the entire system is not limited by its slowest parts (Nielsen and Shumate, 1988:3). To compound the problem, embedded real-time systems must often be implemented to operate under severe hardware constraints due to considerations such as size, power and memory limitations (SofTech, 1986:Ch1, 2). The net result of all of this is the need to pack *more* computation and control into *less* time, space, and processing power.

For obvious reasons, the system must also be extremely reliable. However, its large size guarantees that there will be errors. The software must therefore be designed in a way that minimizes these inevitable errors and tolerates those that remain. Because there will be no opportunity for a full-scale test of the operational defense system, extensive ground-based simulation of battle conditions will be necessary both to detect software errors and inspire confidence in the reliability of the overall system (Eastport Study Group, 1985:12-14). This simulation software will also be extremely complex and computationally intensive, and methods must be found to execute it efficiently on a massive scale.

Although software cost is expected to be a fraction of the total system cost, "the *complexity* of the battle management software to control a given architecture transcends simpler cost measures. Excessively complex software can not be produced at any cost" (Eastport Study Group, 1985:22). This fact demands a departure from past methods of defense software acquisition. "Historically, the pattern used for the acquisition of weapons systems has been to acquire the weapons first and devise the command, control, and communication as an accessory. The Fletcher Report made the point that this 'applique approach' is the wrong approach for SDI" (Eastport Study Group, 1985:5). In short, the expected size and complexity of both the operational and simulation software make their development the focal point of the entire Strategic Defense Initiative.

A successful solution to this seemingly overwhelming problem will require the use of the most advanced hardware and software technologies available. Parallel processing shows great promise not only for meeting the computationally intensive real-time deadlines of SDI software, but also for substituting processing power for software complexity by allowing the use of simple compute-intensive algorithms rather than complex optimized code (Waldrop, 1986:712). The use of parallel processors can increase the reliability of the system by distributing the computing load over many processing units. If one fails or is destroyed, others are available to assume the load (Eastport Study Group, 1985:54). Parallel systems also provide the best alternative for the massive amounts of computing power necessary to execute the ground-based simulations (Eastport Study Group, 1985:50).

In order to exploit the power of parallel processing effectively, it is necessary to have a software programming language which efficiently abstracts parallel algorithms. The Ada language was specifically designed for use in parallel real-time software development. It explicitly supports independent communicating processes through the use of the task and machine-level operations with representation specifications. In addition, Ada is a third generation high-order language with extensive support for software engineering concepts. For these reasons, Ada is an excellent prospect for the SDI software development language.

1.2 *Research Overview*

The purpose of this research is to examine the possibilities presented by parallel processing and the Ada programming language and to determine how they could benefit the development of SDI software systems. Because of the breadth of these topics and the multiplicity of possible applications in the SDI environment, it was necessary to limit the scope of this research to a single representative problem. This problem is used as an example to illustrate the concepts of parallel software engineering in Ada.

1.2.1 The Kalman Filter Tracking Algorithm. A representative real-time problem stems from research into directed energy weapons. Because of the limited power of current lasers, the success of such a weapon would be dependent on a highly accurate tracking system. The Air Force Weapons Laboratory is developing a tracking system which uses a forward looking infrared sensor together with a tracking algorithm to provide target information to the laser pointing system. The purpose of the tracking algorithm is to detect vehicle movement in order to keep the target centered in the tracker's field of view, simultaneously pointing the laser (Tobin, 1986:1-2).

In order to track a target accurately, the tracking algorithm must not only follow vehicle movement, it must also deal with noise such as atmospheric distortions, bending and vibration of the optical hardware, differing target dynamics, and the inherent time lag caused by computation and physical movement of the pointing system. Currently used algorithms do not take these factors into account. A promising tracking algorithm is being developed at AFIT based on Kalman filtering techniques (Maybeck and Mercier, 1980; Maybeck and Rogers, 1983; Maybeck and Suizu, 1985; Tobin and Maybeck, 1987). It has the ability to incorporate all of these factors into the filter dynamics model. As a result, this algorithm has consistently shown an order of magnitude improvement over currently used correlation trackers.

The heart of this tracking system is the Kalman filter. A Kalman filter is nothing more than an optimal recursive data processing algorithm. It processes inputs from measurement devices together with the knowledge of applicable device dynamics, statistical descriptions of noises, measurement errors, modeling uncertainties, and initial conditions,

to produce an optimal estimate of variables of interest. In the case of the tracking system, this variable of interest is the current and predicted future position of the target. The algorithm is recursive in that the optimally estimated variables calculated in one time frame are fed back into the system and used with the incoming measurements from the next time frame to generate new values of estimated variables at the next time instant. For this reason, all of the data from the previous samples does not have to be kept in storage and reprocessed with each new measurement (Maybeck, 1979:4). This is a decided advantage in a real-time system where processing time and storage space are limited.

The tracker can be made even more accurate through the use of a multiple model adaptive filter system (MMAF). The MMAF is composed of a group of Kalman filters constructed with varying target characteristic specifications. The outputs from the filters are arbitrated by a control unit which gives varying weight to each filter, based upon its predictive accuracy as determined by the characteristics of its residual (the difference between the position as given by the measuring devices and that predicted by the filter). The result is an estimate more accurate than a single filter could provide. However, this increased accuracy comes at a cost of increased computational loading. Current research in this area has been devoted to making the filter as accurate as possible with lesser regard to the eventual necessity of processing it in real time. A significant speed-up in the execution time of the Kalman filter tracking system is on objective of this effort.

The selection of this particular research effort as an example real-time problem provides one additional advantage. A large simulation system, which simulates an airborne target track, is necessary to test the efficiency of the Kalman filter tracking algorithm. This system currently consists of a single, sequential FORTRAN program which is extremely time consuming to run. It is an excellent example of the type of ground-based simulation SDI will require on a larger scale. A more time-efficient parallel implementation of this simulation system is also an objective of this thesis.

It is not the intent of this research to improve on the fundamental mathematical concepts of the Kalman filter tracking system or its accompanying simulator. These concepts have been reproduced as faithfully as possible directly from the work of AFIT engineers especially (Tobin, 1986; Norton, 1988). These sources and their accompanying references

should be consulted concerning any specific questions about the tracking algorithm. The Kalman filter tracking algorithm is used only as a relevent example, and the parallel software engineering techniques developed in this thesis can be applied in like manner to any similar software development.

1.2.2 Specific Objectives. This effort has three specific objectives. The first is to synthesize a set of software engineering guidelines which can be used to design a parallel system suitable for implementation in Ada. The parallelization of large, integrated software systems requires complex interaction between the resulting concurrent processes. As a result, parallelization cannot be "applied" to a system after its implementation; it must be built in from the beginning. Therefore, these guidelines must be as complete as possible covering the entire software development process from initial problem analysis to final implementation.

Most of the information currently available on this topic focuses on a particular phase of the analysis or design with little or no guidance as to how that phase fits into the overall development of the project. The primary goal of this thesis is to integrate these ideas into a comprehensive parallel development guide.

The second objective involves proving the validity of these guidelines through the design and implementation of a real-world problem. The Kalman filter tracker research being done at AFIT provides the basis for this validation. The Kalman filter tracking algorithm and its associated simulator will be implemented on a parallel architecture in the Ada programming language. The objective is to produce a meaningful improvement in execution speed over a comparable sequential version.

The final objective of this effort is to examine the adequacy of the Ada language for parallel software development both in real-time and simulation situations. This evaluation will be made based on the results of the Kalman filter tracker and simulator implementations both in objective terms, such as run-time efficiency and accuracy of results, and subjective terms, such as ease of design and coding and support for software engineering principles.

1.3 *Thesis Overview*

This thesis proceeds from a general description of parallel processing concepts to more specific guidelines for parallel software development and finally to a specific example. Chapter 2 consists of background material intended to lay a foundation for understanding this research. It provides an overview of parallel processing concepts, concurrent software concepts, the Ada language, and the basics of the Kalman filter tracking algorithm. Although introductions to both parallel hardware concepts and the Ada language are provided, it is assumed that the reader has some familiarity with both of these topics. If additional background is needed, good introductory material can be found in (Booch, 1983; Nielsen and Shumate, 1988) for the Ada language and (Stone, 1987; Fox et al, 1988) for parallel processing.

Chapter 3 outlines the parallel software engineering guidelines developed to fulfill the first research objective. These concepts are generic to any parallel Ada software system development and can be used independently as a designer's checklist.

Chapter 4 covers the specific analysis and design of the Kalman filter tracking system using the concepts developed in Chapter 3. It follows the software life cycle from its beginning up to the point of coding and can be used as an example for similar software designs.

Chapter 5 is a diary of the implementation and testing of the Kalman filter tracker and simulator. It begins with an overview of the specific hardware architecture and Ada language system used in this research. It continues with a description of the generic routines written to support the tracking system. These routines are common functions with many different applications and are useful in their own right. Following this is an explanation of the steps taken in the coding and testing phases of the project.

Chapter 6 present the research results in terms of the four objectives described earlier and draws conclusions based on these results. It also contains recommendations for further study.

II. Background Research

In any software development, there are three major components: the hardware architecture, the software programming language, and the algorithm to be implemented. The purpose of this chapter is to introduce the reader to these three elements as they apply to this particular research effort. The first section is an overview of parallel processing concepts including a comparison to its architectural rival, a fast sequential machine, and a summary of advantages and disadvantages of this type of computing. The second section begins with a discussion of the demands placed on a programming language by concurrent software development. It proceeds to explain how the Ada language meets these requirements and concludes with the major difficulties in using Ada. The final section presents a basic mathematical description of a Kalman filter and reviews the research done at AFIT on the Kalman filter based tracking algorithm.

2.1 Parallel Processing Concepts

Successful development of a defense against ballistic missiles will require computational resources far more powerful than any in current use. To meet these demands, new computer architecture concepts must be explored. The myriad of new ideas being tested, together with the constant advance of technology in this area, puts the discussion of a particular hardware design beyond the scope of this thesis. Instead, a more general approach was taken, comparing two differing architecture concepts: fast, specialized sequential processors versus parallel processing machines. The reasoning was that selection of the superior concept would result in a superior implementation, relative to the other, without regard to the specific underlying hardware.

The first part of this section is devoted to defining certain key terms associated with the relatively new field of parallel processing. The next subsection compares the salient characteristics of sequential and parallel systems. Following that is a discussion of the advantages of parallel processing. Finally, the corresponding disadvantages are examined and methods are proposed to limit their effect on the implementation of a parallel software system.

2.1.1 Definitions. Significant use of multiprocessor computation, in all its varying forms, is relatively new but expanding at a rapid rate. As in many other fast moving fields, terminology is still inexact and sometimes conflicting. In order to form a common basis of understanding for this thesis, several terms will be defined here.

At the most general level, the use of multiple processors to solve a problem requires that the processors have some means of communicating with one another. Thus, at the heart of this new field is a communications network. The network can connect anything from a group of supercomputers to hundreds of microprocessors, as long as each of them is capable of independent operation. This connection gives the processors the ability to share data, resources, and even work concurrently on a single problem. Simultaneous work on a single problem leads to the next level of abstraction, distributed computing.

By most accounts, distributed processing is a subset of a network, as Tanenbaum explains, "...a distributed system is a special case of a network, one with a high degree of cohesiveness and transparency. In essence, a network may or may not be a distributed system, depending on how it is used" (Tanenbaum, 1981:2). The concept of distributed computing is very broad and includes everything from networked mainframe computers separated by hundreds of miles to individual microprocessors in the same cabinet or even on the same board. The key distinction between distributed processing and a computer network is a control system that makes the group of autonomous computers/processors operate, at a high level, like a single processor. A common operating system eliminates the need for users to know what particular computer/processor they are using, where their files are stored, or even where their programs are running (Tanenbaum and Van Renesse, 1985:419-420).

The next step in abstraction is parallel processing. Parallel processing may or may not be related to distributed computing, depending on how it is used. There are two types of parallelism: local and global. Local parallelism occurs in a single processor, usually in the form of a pipeline. Significant speed-up is possible through the use of instruction and/or arithmetic pipelines that begin the processing of the next operation before completion of the current one (Fox et al, 1988:8-9). Another example of local parallelism is vector processing, where the same operation can be performed on an entire finite size array of

data at once rather than handling each element individually. Local parallelism is not related to distributed computing.

Global parallelism, on the other hand, is one of the possible uses of a distributed system and is a subset of distributed processing. It consists of multiple processors operating concurrently on a single problem. There are two groups of architectures in this category: single instruction stream, multiple data element (SIMD), and multiple instruction stream, multiple data element (MIMD).

SIMD machines run a single program in one processor that controls the operation of all the other processors. Each of these other processors executes the same instruction in lockstep with the others on different data elements. Some degree of control is available through the use of masking to allow certain processors to ignore some instructions, but, in general, each processor will perform the same operation as all the rest (Stone, 1987:279).

In contrast, MIMD architectures consist of independent processors, each capable of executing their own program, that communicate with each other to exchange data and synchronize operations (Stone, 1987:279). MIMD machines can be loosely coupled, with each processor having its own memory and communicating via some type of network, or tightly coupled, with all processors sharing common memory and communicating through that memory. Some hybrid of the two types is also possible. The coupling of the processors generates some design considerations which will be discussed in Chapter 3

Of the two types of global parallelism, MIMD is the most suitable for real-time applications. SIMD machines are best suited for problems that contain repetitive actions on uniform data and where addressing patterns are not data dependent (Stone, 1987:279). Because each processor must execute the same instruction at the same time, this type of architecture does not lend itself well to a real-time system where two processors might have to process data from two different physical sensors, while a third updates an internal database, and a fourth calculates control inputs to feed a pointing mechanism. In an MIMD environment, each processor could perform one of these tasks independently and only communicate with the others when it was necessary to pass information or synchronize some activity.

Review of current literature shows that these definitions are subject to interpretation and overlap in many cases. For purposes of this thesis, any future mention of parallel processing will refer to MIMD systems of individual microprocessors separated by a limited physical distance.

2.1.2 Parallel Versus Sequential Processing. When a single problem requires more computing power, there are two possible alternatives. The first is to build a more powerful single processor. This can be done by speeding up the clock speed so that program steps process faster or by adding more powerful hardware so that complex operations can be executed as a single instruction. It can also be done using methods of local parallelism such as pipelining or vector processing. The miniaturization made possible by VLSI technology has made for a long and profitable run in all of these areas, but single processors are reaching the limits of their physical capabilities. David Shaw, director of the NonVon Supercomputer Project at Columbia University explains, "We are running up against fundamental physical barriers in designing von Neumann machines. Traditional supercomputer architects are now dealing with speed-of-light limitations and exotic cooling technologies, but their efforts are reaching the point of diminishing returns" (Livingston, 1985:26).

If single processors have reached the limits of their capabilities, the remaining alternative is a parallel processing system. Given forseeable technology, this is the only way to guarantee the ability to perform complex computation in real time. Stone explains, "the motivation for moving towards multiple processors is strictly a matter of performance because device technology places an upper bound on the speed of any single processor" (Stone, 1987:278). Fox et al agree, "There is now general agreement that the only route to significantly increased performance is through concurrent computation – the use of many computers together to solve the same problem" (Fox et al, 1988:2).

Theoretically, the potential for performance increase is unbounded. As the problem gets larger, one simply adds more processors. In reality, limiting factors such as control and communciations overhead will reduce this performance, but the overall message is clear; parallel processing is the best way we now have to meet our burgeoning computational needs.

The increased speed of parallel processing holds special advantages for real-time systems. Because frame times are normally fixed, increased speed means spare computation time. Spare computation time can be used to implement more complex, accurate control systems, or it can be used to increase software maintainability by allowing a more "straightforward implementation" or even the use of a high-order language (Westermeier, 1981:1010-1011).

Increased processing speed also provides the opportunity to shorten frame times. This may make it possible to reduce lag times between data sampling and the propagation of the corresponding control signals. Shorter frame times can also be used to increase the data sampling rate resulting in better data filtering and minimization of the effects of aliasing (Westermeier, 1981:1011).

2.1.3 Advantages of Parallel Systems. In addition to superior performance, the use of parallel computing has other advantages as well. One that is critically important to real-time systems is increased reliability. With a single-processor system, hardware failure is fatal. The best that can be hoped for is graceful degradation through the saving of current state information. With catastrophic failure, not even this is available. A parallel system increases reliability through simple redundancy. If one processor fails, its workload can be shifted to another processor. Even if specific workloads cannot be shifted, the system can still continue to operate in a degraded mode (Knight and Urquhart, 1983).

Another advantage is the possibility of incremental growth. Complex military systems must constantly change to counter new threats or incorporate new technology. In many cases, these new changes will require more computing power. When the capabilities of a single processor are exceeded, the entire system must be replaced. This often requires major changes to the software. With a parallel system, increased capability can be added with additional processors. In many cases, this incremental growth can be achieved with a minimum of hardware or software redesign (Lane and others, 1983:327).

A final advantage is a decrease in cost. This is especially crucial in a large system to minimize recurring hardware costs both for the initial system and any replacement parts. High-speed, specialized processors require customized VLSI design and sophisticated

support systems. Parallel systems can be constructed from relatively cheap, mass-produced processors. The relatively slow step times in these processors minimizes heat dissipation and transmission delay problems (Lerner, 1985:22). In general, parallel systems can claim a price/performance advantage over traditional systems (Tanenbaum and Van Renesse, 1985:420).

2.1.4 Problems to be Overcome. Despite its advantages, parallel processing is not a panacea. In general, parallel systems present design considerations that do not exist in their single-processor counterparts. Performance is reduced by inefficiencies such as communication between processors. Message passing takes time and processing power and may lead to additional delays if processes must wait for synchronization. Another source of overhead is the additional operating and run-time system control necessary for task scheduling and management. These operations are far more complex than in a single processor machine. A final problem can come in the form of improper load balancing. This causes some processors to become idle while others may thrash from too much workload (Stone, 1987:281).

In the real-time embedded systems arena, parallel processing leads to additional considerations. Increased hardware and software overhead means increased size, weight, and power requirements for the system. In addition, the added complexity of parallel programs may reduce system reliability. The use of microprocessors reduces the effect of the first problem, but the latter can only be limited through effective design.

These inefficiencies can never be completely eliminated, but they can be significantly reduced. The use of proper partitioning and design methods will lead to favorable load balancing. Such methods will be discussed further in Chapter 3. In addition, the proper use of a programming language with specific concurrent processing constructs can reduce the overhead of communication and task management. Ease of design and quality of implementation will be directly dependent on the language selected.

2.2 *Concurrent Software and the Ada Language*

The programming language is the tool with which a programming team abstracts a problem into its software representation. If this tool is specifically calibrated for the problem at hand, it can significantly ease the translation from the real world to a computer model. If not, the efficiency of the software developers can be severely reduced, and the resulting program can be difficult to understand and maintain. For this research, the language selected must support the development of real-time, concurrent software in as efficient a manner as possible, consistent with the principles of software engineering.

After a review of the principles of software engineering, this section outlines the specific requirements placed on a programming language by concurrent software development. The additional requirements of real-time systems are also discussed. The next subsection explains how the Ada language meets the requirements of software engineering, concurrent software development, and real-time systems. The final subsection outlines the problems that must be dealt with when using Ada.

2.2.1 Principles of Software Engineering. When programs were relatively simple, they were often developed in a haphazard fashion according to the programmer's intuition. The results were acceptable as long as the program was small enough for a single programmer to complete, the programmer was good at what he did, and no one else ever had to maintain the code he produced. With today's more complex systems, this method is no longer acceptable. Programs have become too large for one person to write alone, many people in the industry lack extensive understanding of computer science, and maintenance has become a nightmare. The result is what is popularly known as *The Software Crisis*. This is a crisis marked by cost and schedule overruns, software that is unresponsive to user's needs, unreliable in what it does do, and very difficult to modify.

Out of this chaos grew the concept of software engineering, an attempt to apply standard engineering practices to what was formerly a "black art." Several principles have been advanced, the use of which will increase the efficiency of the programmer and the quality of his product. These include: abstraction, information hiding, modularity, localization, uniformity, completeness, and confirmability. The following discussion has

been abstracted from (Booch 1983:27-30).

Abstraction is the process of taking a complex problem and decomposing it into its simpler components. This process continues until the individual components are simple enough to be reliably implemented. Abstraction is especially important for very large systems or those whose complexity would otherwise be difficult to comprehend, such as concurrent programs. Information hiding is the other side of the same concept. At each level of abstraction, any information that is not necessary to understand that level should be inaccessible to the developer. For example, if the programmer needs to use a stack, all that should be of interest to him is the available stack operations, not information on how the stack is implemented.

Two other related ideas are modularity and localization. Modularity is the result of implementing abstractions. Each component created by decomposition should be modularized into either a function module specifying a group of operations or a declarative module specifying a group of objects. Experience suggests that modules should exhibit loose coupling, a measure of interconnection between modules, and tight cohesion, a measure of the relation of the modules internal elements. In simpler terms, a module should perform a single specific purpose and only interface with the outside world where absolutely necessary. Localization helps create modules with these properties. Data elements and operations local to the module are made inaccessible to the outside world.

Completeness and confirmability relate to the adequacy of a single language to implement a problem. A language should be complete so that calls to the operating system or routines written in other languages will not be necessary. Languages should also have features that assist the developer in finding any errors in the program. An example of such a feature is strong typing. This allows the compiler to identify possible errors in variable usage. Uniformity comes with a language where modules are specified in a consistent style.

2.2.2 Additional Requirements of Concurrent Software. One of the most common models suggested for allocating software across a parallel system is a collection of autonomous communicating processes. The interactions between the processes are defined independent of their distribution. The processes are not under central control, and, when

not communicating, they are capable of operating in parallel. The hardware configuration is kept transparent to interprocess communication by passing messages between processes (Lane et al, 1983:327).

The concept of autonomous communicating processes extracts the maximum benefit from parallel hardware by minimizing overhead and maximizing independent processing. It is also abstract enough to simplify the software design process. To use this model, there are several requirements that must be met either by the programming language, the executive system, or explicitly by the programmer at a very low level. The ideal situation is for the programming language to handle these details, freeing the developer to concentrate on the high-level design.

First, the language must provide some construct for specifying independent processes. Once the processes are running in parallel, a method is needed to synchronize them to allow for the exchange of information. Communication should occur only at strictly controlled points using well-defined interfaces. Independent processes should not have unrestricted access to each other's operations (Lane et al, 1983:327). Not only must operations be protected, but data as well. Some method for mutual exclusion must be provided to prevent the simultaneous access of data at processor interfaces (Mundie and Fisher, 1986:21). Finally, parallel processors are inherently non-deterministic. The language must provide some method of controlling this non-determinism in the cases where a proper sequence of events is necessary to ensure the correct result (Lane et al, 1983:327).

In addition to these conceptual issues, there are some run-time requirements that parallel languages must meet. "Traditional real-time systems are designed with the assumption that the primitives required for concurrent operations are provided by a real-time executive ..." (Nielsen and Shumate, 1987:695). This real-time executive must handle the creation, scheduling, and dispatching of processes. It must also manage process communication including queue and buffer management (Nielsen and Shumate, 1988:5-6). The quality of the executive, and the level of abstraction of its interfaces, determine the amount of effort required by the programmer to accomplish these low-level activities.

A final important issue, which embodies many of the concepts already discussed, is

the degree to which the language shields the programmer from the underlying hardware. The developer should not be forced to learn the physical details of the hardware in order to write the software. System dependence decreases maintainability and transportability. "Hence, a language should have a flexible and general method of specifying interprocessor communications which hides the details fo the network from the user, as well as providing a means of access to the network directly if desired" (Cline and Siegel, 1985:971).

2.2.3 Additional Requirements of Real-time Systems. Real-time systems have their own requirements in addition to those necessary for parallel systems. The language must have some method of communicating directly with peripheral hardware such as sensors and control devices. Included in this is the need for interrupt handling and direct input and output. If this capability is not available, the result will be more assembly code inserts.

A real-time language must also be efficient because of the tight time and space constraints common to embedded systems. Finally, the language must help the software engineer meet the demanding reliability standards necessary for real-time systems. It can do this in three ways: by helping to minimize errors in production, by increasing the ease with which errors are found in subsequent testing, and by providing some mechanism for dealing with the unforseen errors that will inevitably appear after the system is in operation.

2.2.4 Adequacy of the Ada Language. The Ada language was born out of the software crisis and an acknowledged need for a high-order language to use in programming embedded systems. In the past, "...the inefficiency of compilers has often meant that HOL's were not feasible or cost effective for embedded software applications" (Westermeier and Hansen, 1985:597). Under the guidance of the DoD High-Order Language Working Group, a slow, deliberate development process led to a language superior in its support for software engineering concepts, concurrent software development, and real-time systems.

Ada has numerous facilities to support software engineering concepts. Some of them such as strong typing, subprograms, and a wealth of control structures are available in some other high-order languages such as Pascal. However, some Ada-unique constructs are of

particular interest. Abstraction is supported by encapsulating data functional resources in the package construct. The interfaces to a package are strictly defined and controlled in the package specification. The actual computational resources are contained in the package body and are inaccessible to the user. This also supports the idea of information hiding, as does the concept of private types (Bolz, 1986:3).

Modularity and localization are supported by several Ada constructs including the package, procedure, function, and generic. The generic is another Ada-unique construct that greatly increases productivity by increasing the reusability of code. A generic can be a package, procedure or function. It specifies the template of a general algorithm where functions or data elements that change are left out. This template can be used repeatedly by specifying those missing parts and creating an entirely new module. In addition, Ada is a very uniform language. All of these constructs have similar structures and interfaces.

The language is very complete. It contains not only the normal structures expected for sequential computation, but also specific support for parallel processing (the task), error handling (the exception), and direct interface with hardware devices (the representation specification). As a result, the programmer must seldom, if ever, go outside the confines of the language to complete his task.

Ada has explicit support for parallel processing. Tasks and packages are used to produce individual processes with controlled interfaces and internal invisibility to the outside world (Lane et al, 1983:327). The task rendezvous provides for synchronization of processes by forcing the sender to wait until the receiver is ready before transmitting information. The *entry* call provides a method of communication and prevents common access of a system resource (Mundie and Fisher, 1986:21). Finally, the use of *select* statements allows the programmer to deal with the timing problems and non-determinism inherent in parallel programming (Lane et al, 1983:327). A more detailed explanation of tasking features is provided in Chapter 3.

The Ada run-time system handles all task activation, scheduling, dispatching, and deactivation. It also manages task communication regulated by the *entry* calls and *accept* statements in the communicating tasks. This provides a very high level of interface

for the programmer and eliminates the need for a real-time executive (Nielsen and Shumate, 1987:695). "The Ada language enhances transportability by eliminating almost all dependence on an operating system. Hence, such capabilities as exception handling and concurrency become programming language features as opposed to operating system features" (Bolz, 1986:1).

Ada also provides extensive support for real-time specific needs. Low-level communication is available at the machine level through the use of representation specifications. This eliminates the need to write assembly code interfaces to accomplish these tasks. Reliability is enhanced in original production by the software engineering support in the language, in testing by the ease with which Ada code is understood and changed, and in the operational phase by the use of exception handling. The efficiency of the language is, of course, dependent on individual compiler and run-time system implementations, but good results have already been recorded. A distributed flight control system for the F-15 was developed in Ada with only a 10% increase in execution time and a 36% increase in memory usage over its assembly language equivalent (Westermeier and Hansen, 1985:603).

2.2.5 Potential Problems with the Use of Ada. Despite the fact that Ada was specifically designed for use in a parallel, real-time environment, early research has brought several potential problems to light. To begin with, many parallel processing considerations are not covered by the Ada standard including: "the allocation of processors: static, dynamic, user-defined, or automatic; the atomicity of distribution: packages, tasks, or procedures; possible remote operations: rendezvous, activation/termination, remote procedure calls, and global variables; remote dependencies and exception handling; and general network topics: encryption, protocols, fault handling, and so forth" (Paulk, 1986).

Another problem is that the parallel features of the Ada language, as currently defined, seem to imply a tightly-coupled system. The language definition does not provide explicit tools for connection management in a loosely-coupled environment. There is no construct available to represent a node in a distributed network, specify a communications path, or explicitly bind a program unit to a particular processor. There is also no specified method of memory management for the use of global variables (Paulk, 1986).

Despite this lack of explicit support, it is still possible to handle them using the run-time system. A method for this purpose, called Distributed Ada Tasking, has been considered by (Rourk et al, 1987). This scheme allows Ada programs to be distributed across multiple processors with the task serving as the boundary unit. All communication between tasks is handled via the rendezvous. They conclude that the technology currently exists to implement such a run-time system in a loosely-coupled multiprocessor environment.

One constraint of the Ada task construct is that it does not support asynchronous communication. A rendezvous is always blocking for either the caller or called task (depending on whether the *entry* or *accept* statement is reached first) (Paulk, 1986). This is a disadvantage in the case where a designer may want a particular process to simply leave a message and continue processing. The inadequacy can be overcome using the concept of a buffering task. This task's only purpose is to buffer messages from a sending task until the intended receiving task is ready to accept them. This concept will be explained more fully in Chapter 3.

A fourth problem with Ada is its lack of explicit hardware fault-tolerance support. Because of the synchronous nature of task communication, the task construct is vulnerable to hardware failure (such as a processor or network crash) during certain periods of the rendezvous. For example, if a server task is lost after a calling task has made an *entry* call, the calling task will remain suspended forever, waiting for the server to respond. Research into this area of fault tolerance has produced some promising results using timed calls and selective calls (Reynolds et al, 1983), but more work needs to be done.

A final area of difficulty is the production of an Ada run-time system efficient enough for real-time operations. The Ada run-time system is a significant help to the software engineer, freeing him from having to deal with difficult task management problems such as allocation, scheduling, and communication. However, the run-time system can be slow and cumbersome, especially in the areas of task support, memory management, and exception handling. These systems must efficient if they are to function adequately in a real-time environment. One possible solution to this problem is to develop hardware support for each of these functions (Kamrad, 1984). For information on progress in this area, consult

(Justin and Lattin, 1981; Caruso, 1985; Booch, 1986).

In all fairness to the Ada language and its designers, most of these problems do not exist in other languages primarily because parallel processing support does not exist in other languages. As the first widely used language to provide such support, Ada is undergoing understandable growing pains in this area. Yet, none of these problems are insurmountable and most can be circumvented using good design technique.

2.3 Tracking Algorithm

A tracking algorithm in the SDI environment must meet two criteria. First, it must produce as accurate a tracking response as possible. This is necessary because of the high speed and maneuverability of airborne targets and the extremely precise aiming required by the laser weapon. Second, it has to present a reasonable amount of computational loading in order for the system to run within real-time constraints. Highly accurate algorithms are of little value if processing them takes more time than is available in the frame. A tracking algorithm based on Kalman filtering techniques can be shown to meet both of these criteria.

This section begins by presenting the basic mathematics involved in the Kalman filter. Following this is a short summary of the research done at AFIT in the use of Kalman filters for tracking algorithms. That research forms the basis of the example implementation in this thesis. The final subsection explains the advantages of a Kalman filter design.

2.3.1 Kalman Filter Basics. As stated earlier, a Kalman filter is an optimal recursive data processing algorithm. It is optimal in the sense that error is minimized statistically; the average results obtained from a Kalman filter will be better, using almost any criteria, than the average results of any other type of filter. This optimality is dependent on three assumptions.

The first is that it must be possible to describe the system of interest with a linear model. Second, the noise in both the system and the measurements must be "white." This implies that noise value is uncorrelated in time and has equal power spectral density value at all frequencies. This assumption is necessary to make the math tractable. Finally,

the amplitude of this noise must be Gaussian. This assumption makes it possible for the Kalman filter to propagate all information contained in the conditional probability density for state variables, conditioned on all measurements seen, through the first and second order statistics. While these assumptions obviously do not describe the real world, a convincing case for them is made in (Maybeck, 1979). It is also possible in many applications to relax these restrictions when necessary and still retain most of the benefit of the filter.

The following simple example of a sampled-data Kalman filter is taken from (Maybeck, 1979) and (Maybeck, 1988). Assume that the next state of a certain linear system can be described by the equation:

$$x(t_{i+1}) = \Phi x(t_i) + Bu(t_i) + Gw(t_i) \tag{1}$$

where:

x = system state

Φ = state transition matrix

B = control input matrix

u = control

G = noise input matrix

w = dynamics white noise with mean 0 and covariance matrix Q

and the measurements to be input into the system can be described by the equation:

$$z(t_i) = Hx(t_i) + v(t_i) \tag{2}$$

where:

z = measurement

H = measurement matrix

v = measurement noise with mean 0 and covariance R

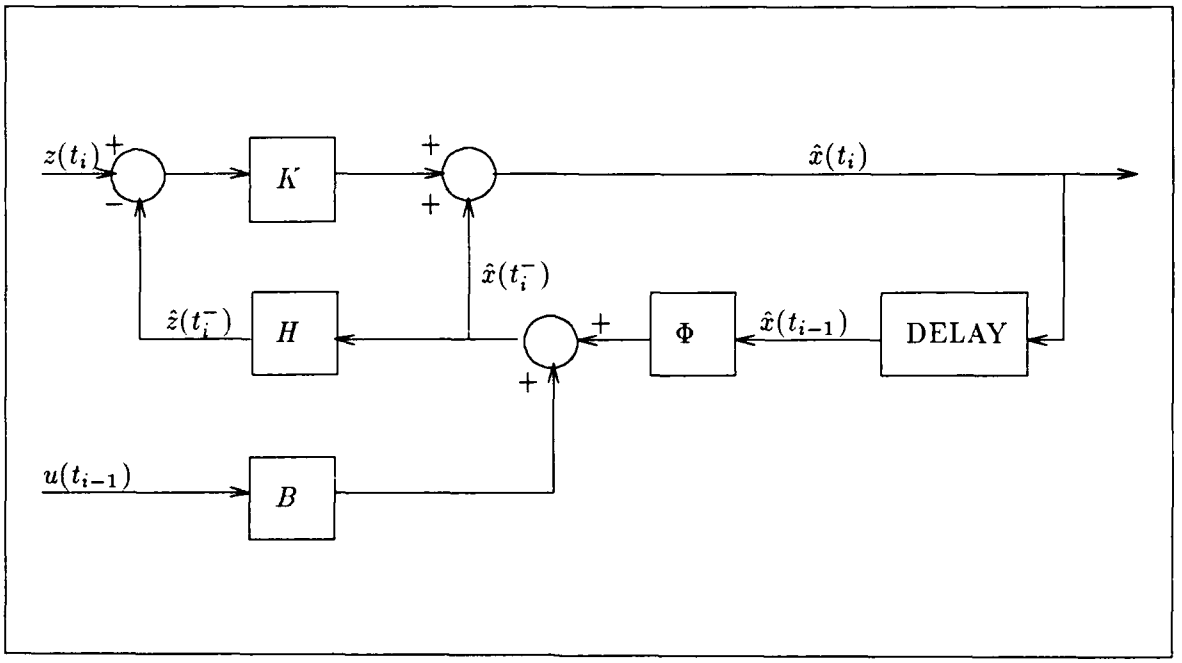

Figure 1. Sampled Data Kalman Filter Block Diagram (Maybeck, 1988)

In addition, $P(t_i)$ is the covariance of the error of the estimate of x and will be determined later. The block diagram for the resulting Kalman filter based upon this model is shown in Figure 1.

The new measurement $z(t_i)$ enters the system and the filter's best estimate of what it should have been is subtracted from it. This prediction was propagated from the previous best estimate of the system state $\hat{x}(t_{i-1}^+)$, where "+" denotes after the measurement at that time is incorporated, using the following equations:

$$\hat{x}(t_i^-) = \Phi\hat{x}(t_{i-1}^+) + Bu(t_{i-1}) \tag{3}$$

$$P(t_i^-) = \Phi P(t_{i-1}^+)\Phi^T + GQG^T \tag{4}$$

where "−" denotes before the measurement at time t_i is incorporated. This is projected by the measurement matrix H into a prediction of the measurement's variables. The difference between the measurement $z(t_i)$ and the filter's prediction of what it should be is known as the residual $r(t_i)$ and is calculated as:

$$r(t_i) = z(t_i^-) - H\hat{x}(t_i^-) \tag{5}$$

The residual is then passed through a weighting matrix $K(t_i)$, also known as the Kalman filter gain, which adjusts it based on the current confidence in the internal filter dynamics model relative to the current confidence in the output of the measurement devices. The Kalman filter gain is determined using the equation:

$$K(t_i) = P(t_i^-)H^T[HP(t_i^-)H^T + R]^{-1} \tag{6}$$

High confidence in the measurement devices (as expressed through small R values) will lead to a larger $K(t_i)$, while higher confidence in the filter's internal dynamics model will mean a smaller $K(t_i)$.

The weighted measurement residual is then combined with the filter's best prediction of the systems' estimated state at the instant before the measurement was taken. The result is the new best estimate of the current state and is described by the equations:

$$\hat{x}(t_i^+) = \hat{x}(t_i^-) + K(t_i)[z(t_i) - H\hat{x}(t_i^-)] \tag{7}$$

$$P(t_i^+) = P(t_i^-) - K(t_i)HP(t_i^-) \tag{8}$$

The basic model just described is for a linear system. A nonlinear system can be modeled with an extended Kalman filter (EKF). It contains the same sequence of propagation and update cycles. "A complete EKF derivation involves writing the nonlinear system dynamics and measurement equations in a Taylor series expanded about the most recent value of the state estimate" (Tobin, 1986:18). These state equations are made "linear" by eliminating all terms above the first order. As a result, the extended Kalman filter is no longer optimal. However, the use of this model may be necessary in certain situations in which system dynamics and/or measuring devices exhibit non- negligible nonlinear characteristics.

2.3.2 Summary of AFIT Research. Research has been conducted at the Air Force Institute of Technology on the use of Kalman filtering techniques for tracking algorithms since 1978. This summary of that research is abstracted from (Tobin, 1986) and (Tobin and Maybeck, 1987). Consult these references for the individual sources of research.

Initial work in this area showed the usefulness of an extended Kalman filter algorithm for tracking long-range point targets based on forward looking infra-red (FLIR) measurements of intensity over an 8-by-8 array of picture elements (pixels). A four-state filter was used to model position in two directions (the two dimensions of the FLIR image plane) and atmospheric jitter in the same two directions. Follow-on research increased the accuracy of this filter by adding states for velocity and acceleration of the target. The system was also adjusted to account for target shape effects so that it would no longer be dependent on the point target assumption.

A significant addition to the original work was the development of an enhanced correlation algorithm to be used in series with a linear Kalman filter. The correlator "preprocesses" the input from the forward looking infrared sensor by comparing it with a target shape template, constructed by the use of a finite memory image averaging technique (using the filter to assist in centering each image before the averaging occurs). The resulting position information output by the correlator is in the form of a linear offset from the center of the field of view. The information can therefore be fed directly into a linear Kalman filter. The cascaded algorithms cause considerably less computational loading than a corresponding extended Kalman filter design.

Up to this point of development, the algorithm performed extremely well for relatively benign (slowly maneuvering) targets, but could not adequately track target maneuvers in excess of 5 g's. A different approach was needed to handle targets that could display a wide range and rapid variation of maneuvering characteristics, by allowing rapid change in the bandwidth and field of view of the tracker. The result is the multiple model adaptive filter (MMAF). The block diagram for this filter is shown in Figure 2.

An MMAF for this application consists of a group of independent Kalman filters, each of which is tuned for a different model of expected target behavior. In Figure 2, the values $a_1, a_2, \ldots, a_k$ correspond to specific assumptions about the model upon which that filter is based, expressed in terms of discrete values that model-defining parameters can assume. Incoming measurements are fed into each of the filters in parallel. The resulting residuals from all of the filters are fed into a hypothesis conditional probability computation algorithm, to evaluate the probabilities that each of the elemental filters is the "best" one

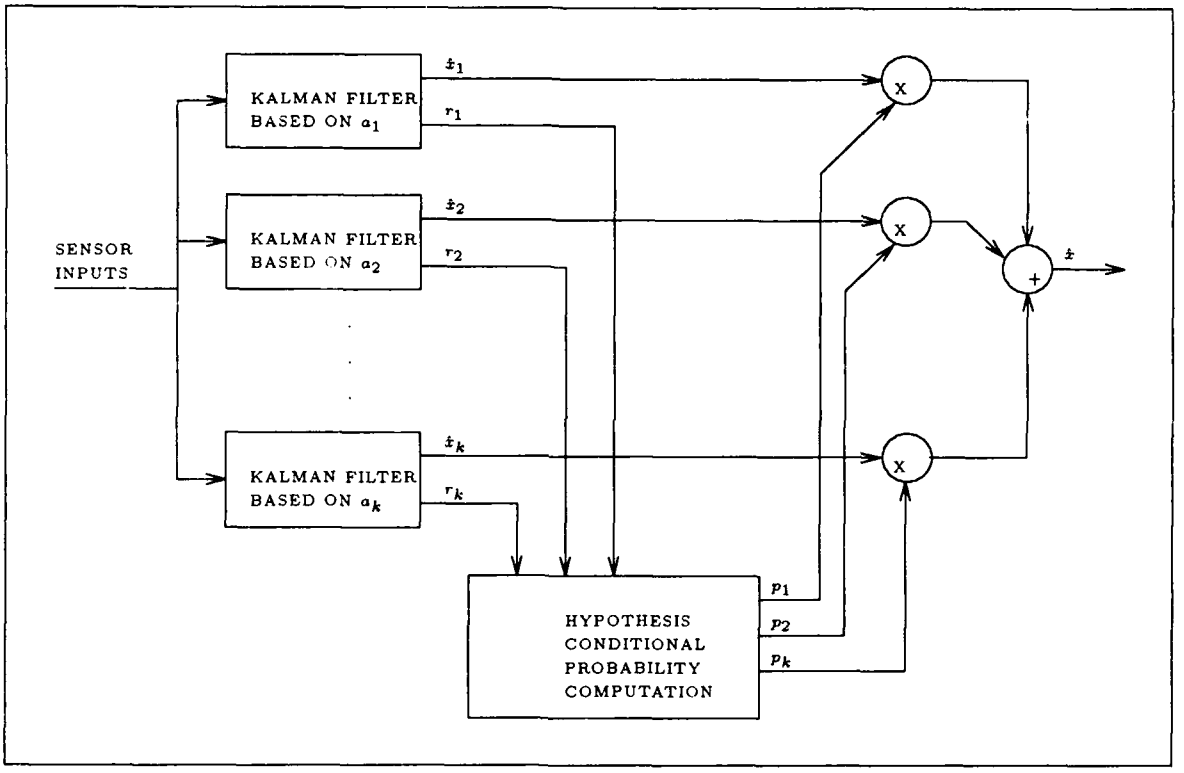

Figure 2. Multiple Model Adaptive Filtering Algorithm (Tobin and Maybeck, 1987)

to use to track the target currently, conditional on the history of measurements seen. The reasoning is that the filter with the residual most in consonance with the filter-computed residual covariance of $[H P(t_i^-) H^T + R]$ (see Equation (6)) will be the one using a dynamics model closest to the actual track of the target and should be the most heavily weighted. The output from this algorithm is a weighting scheme that combines the current state outputs from all of the filters into a single result.

Two methods have been tested for producing the final state estimate. The first, known as the Bayesian MMAF, generates a probabalistically weighted average of the outputs from all of the filters, as displayed in Figure 2. The second, the Maximum A Posteriori approach, does not use the state outputs of all the filters. It simply selects the output of the filter with the highest computed probability. In both cases, the result is a tracker able to handle very dynamic targets without sacrificing high resolution for benign trajectories. In either case, the hypothesis conditional probability is calculated using the following equation:

$$P_k(t_i) = \frac{f_{z(t_i)|a,Z(t_{i-1})}(Z_i|a_k, Z_{i-1}) P_k(t_{i-1})}{\sum_{j=1}^{J} f_{z(t_i)|a,Z(t_{i-1})}(Z_i|a_k, Z_{i-1}) P_k(t_{i-1})} \tag{9}$$

where the density that appears in the equation is described as:

$$f_{z(t_i)|a,Z(t_{i-1})}(Z_i|a_k, Z_{i-1}) = \frac{exp[-1/2 r_k^T(t_i) A_k^{-1}(t_i) r_k(t_i)]}{(2\pi)^{m/2} |A_k(t_i^-)|^{1/2}} \tag{10}$$

where $A_k(t_i)$ is generated in the kth Kalman filter as:

$$A_k(t_i) = H_k(t_i) P_k(t_i^-) H_k^T(t_i) + R_k(t_i) \tag{11}$$

2.3.3 Advantages of a Kalman Filter Based Algorithm. The Kalman filter based tracker meets both of the criteria presented at the beginning of this section. The algorithm is accurate, having shown an order of magnitude improvement in tracking performance over currently used correlation trackers. In addition, the recursive nature of the filter makes it unnecessary to store and reprocess previous measurement information after each new measurement is taken. This results in a large reduction in computational loading and

storage requirements. On-line computing requirements can be reduced even further because of the separability of the mean and covariance equations. This allows the covariances and optimal gains of each of the elemental filters in Figure 2 (Equations (4), (6), and (8)) to be precomputed and stored.

2.4 Chapter Summary

The three major components of this research are a parallel multiprocessor hardware architecture, the Ada programming language, and a Kalman filter based tracking algorithm. Parallel processing was chosen because it provides the most promise for meeting the massive computational needs of SDI simulation and real-time systems. In addition, the use of parallel machines can result in increased reliability, the possibility of incremental growth, and an overall decrease in system cost.

The Ada programming language was selected because of its explicit support for embedded real-time systems including specifically designed constructs for: parallel processing (the task), error handling (the exception), and direct interface with hardware devices (the representation specification). In addition, Ada is an advanced third generation high-order language with extensive support for modern software engineering practices.

The Kalman filter based tracking algorithm was selected as an example problem because it, and its associated simulation program, are representative of the types of software developments SDI will encounter on a larger scale. In addition, the multiple model adaptive Kalman filter system being studied at AFIT is a superior tracking algorithm both in terms of accuracy and computational loading.

III. Parallel Software Engineering Guidelines

This chapter offers a comprehensive set of guidelines for parallel analysis and design with an eye towards final implementation in the Ada language. The first three sections are concerned with problem analysis and the last three with system design. The first step is to decompose a problem into independent components which can be parallelized. Section 1 explains three methods for doing this. Once the problem is decomposed, the individual processes must be mapped to a corresponding construct in the implementation language. Section 2 outlines the different types and levels of partitioning and explains why the task is the best level for Ada. Section 3 is an overview of the features of Ada tasks. Section 4 describes a new method of real-time process control called data driven design. Section 5 applies the software engineering principles described in Chapter 2 to a concurrent software environment using the task as a basis. Finally, Section 6 outlines a methodology suited to the design of concurrent software in Ada.

3.1 Problem Decomposition

The first step in any parallel software project is to determine how the problem can be decomposed into independent processes. This section describes three methods of problem decomposition. The terminology developed in (Quinn, 1986) is used to provide a common basis for definition. The three methods are: relaxation, pipelining, and partitioning. Following a description of these methods is a discussion of the issues affecting decomposition, including the factors that limit parallel efficiency and the objectives of a good decomposition.

3.1.1 Methods of Problem Decomposition. Westermeier identifies two methods of problem decomposition for real-time systems (Westermeier, 1981). One method consists of dividing processing into independent functions. Each of the resulting processes operates on the same input data and performs a complete function. Quinn calls this type of decomposition relaxation because the processes are not dependent on data inputs from each other, and no synchronization is required between them (Quinn, 1986:60).

This type of decomposition is ideal when multiple tasks are being performed in the same system at the same time. This is a common occurrence in real-time systems. Relating the resulting algorithm to the single processor case, "each of the parallel processors executes the mainstream functions of the uniprocessor, but in smaller amounts" (Westermeier, 1981:1014). Given a constant sample rate, this type of parallelization will result in a shorter frame time.

The other method proposed by Westermeier consists of dividing the problem into functions that follow each other sequentially. Each process performs its portion of the computation on data received from the one before it and passes the results to the one that follows it. Quinn calls this pipelining and emphasizes that all segments must produce results at the same rate or the system will bottleneck at the slowest process (Quinn, 1986:58-59).

Pipelining works well for problems where complex operations must follow each other sequentially. The sequential nature makes it impossible to decrease the overall frame time because a single data set will still require the same overall amount of processing time. However, once the pipeline is full, results will emerge at a faster rate, making it possible to increase the sample rate (Westermeier, 1981:1013).

There is also a third method of problem decomposition. Quinn calls it partitioning. Instead of each processor performing different computations, a group of processors works simultaneously on subparts of the same problem. The results are combined in a final step to form the solution (Quinn, 1986:59). This method differs from the other two in that partitioning is done by dividing the problem data domain rather than its algorithmic functions. The result of the divided data domain is called the grain, and grain size has an important effect on the efficiency of the resulting code (Fox et al, 1988:43).

Partitioning is most effective for homogeneous operations performed on relatively large data sets. Typically these can be identified through iterative structures. Stone explains, "in typical applications the bulk of the speedup obtained from parallelism is through the parallel execution of loop iterations" (Stone, 1987:375). He provides a method for identifying areas of a program where partitioning can be used. First, find the loop or recursion

structures in the program. Find the instructions that account for the most processor time and split the regions across n processors. Finally, add the necessary synchronization and data transfer to complete the problem (Stone, 1987:334).

In general, there are three methods of decomposing software for parallel computation. Relaxed algorithms can be used for asynchronous operations of processes that do not depend on data from one another. Pipelined algorithms are used in those cases where a problem can be divided functionally into separate blocks, each of which receives data from the previous block, processes it, and passes it on to the one that follows. Finally, partitioning can be used for problems where homogeneous operations are performed on large data sets. Each of these methods will result in speedup over sequential equivalents if used in the proper contexts, but they are all subject to the decomposition limitations described in the next section.

3.1.2 Decomposition Issues. There are three major limiting factors that impact the decomposition of a problem and the efficiency of parallel software. The first, and in most cases the most significant, is the computational overhead added by parallel software. This overhead comes in one of three forms: tasking, communication, and synchronization.

Tasking overhead is a function of the number of independent processes in the system and the efficiency of the run-time environment. It consists of "the work required to generate the task, to enqueue it while waiting for a processor, to dequeue it when a processor becomes available, and to log the completion of the task so that some barrier can be passed when all tasks are completed" (Stone, 1987:338).

Communication overhead is a function of the amount of data that is passed between individual processes and the efficiency of the communications medium. Communications overhead fits into one of two categories. The first is that which is inherent in the programming language. In Ada, this includes the computational costs of subprogram invocation, task rendezvous, task activation/termination, and data reference/modification (Chow and Feridun, 1986:10-12). These costs are relatively constant throughout the execution of the program.

The second type of communication overhead is the computational cost and time

delay incurred by physical message passing. This is essentially nil on a shared memory machine, but can be very significant on a loosely coupled system. It includes: "virtual line time, queueing at the port queues on the route in a store and forward network, and the processing, waiting, and queueing time of the corresponding process and its processor" (Williams, 1983:404).

Finally, synchronization overhead is a function of the number of times individual processes must suspend operations in order to communicate with other processes and the amount of time they spend in that suspended state. The result is idle processor time. The efficiency of the load balancing has the greatest effect on this type of overhead.

The second limiting factor in parallel speedup is the percentage of sequential operations in the problem that cannot be parallelized in any manner (Quinn 1986:61). These sequential operations must be performed on a single processor while others are held idle. This reduces the efficiency of the overall algorithm. It also adds significantly to synchronization overhead as other processors are locked waiting for the completion of sequential code.

The final limiting factor is contention for data. This is a result of several processors requiring access to the same global data element at the same time (Quinn, 1986:61). On a shared memory machine, processes queue up to obtain the data, resulting in a serialization of the parallel tasks. On a loosely coupled system, data contention results in increased message passing and increases the possibility of multiple copies of the same global variable existing in different states. This effect is called software lockout and must be minimized whenever possible.

With these limiting factors in mind, Williams suggests four objectives that must be met to perform effective problem decomposition. The first is to minimize interprocess communication. This objective can be partially met by clustering functions that communicate frequently into a single task (Williams, 1983:404-405). Additionally, careful partitioning and assignment of data structures will reduce the need for data transfers.

The second objective is load balancing. Here, task allocation is critical. Tasks must be scheduled to ensure the overall computational load on each processor is as equal

as possible. Two types of task allocation exist: static decomposition, where tasks are generated and scheduled before execution, and dynamic decomposition, where tasks are generated by the program during execution. In general, dynamic decomposition allows for more efficient task scheduling (Quinn, 1986:61-62). Efficient static decomposition is possible using a partitioning method such as that developed by (Chow and Feridun, 1986).

The third objective is to ensure that each processor meets minimum processing requirements. The grain size of each partition must be sufficient to warrant its inherent parallel overhead. Also, processors must have access to sufficient processes so that there will always be one available to execute while others are blocked waiting on messages (Williams, 1983:405).

The final objective applies only to loosely coupled architectures. In the case where significant physical distance separates processors or where network limitations result in having to pass messages though intermediary processors, tasks with communications requirements should be allocated as close as possible to those with which they communicate (Williams, 1983:405).

The degree to which these objectives are met will determine the overall efficiency of the parallel algorithm. Because of these additional considerations, the fastest sequential algorithm is not always the best candidate for parallelization. Often slower sequential algorithms will lend themselves to better decomposition (Quinn, 1986:62). With these decomposition objectives in mind, the next section evaluates the various levels at which the Kalman filter tracking algorithm can be decomposed and what the most appropriate method is at each level.

3.2 General Partitioning Strategy

Once the problem is decomposed, the designer is left with a group of problem fragments. Each of these fragments can be performed independently and may communicates with other fragments at specified times. The next step is to map these fragments into software constructs suitable for parallel operation. This is known as partitioning. The type of constructs available for partitioning will be dependent on the programming language

selected for the implementation. This section discusses the options available for partitioning and explains why using the pre-partitioning strategy with the Ada task is the superior method of partitioning in Ada-based designs.

3.2.1 When to Partition. There are two times in the development process when software can be partitioned. The two corresponding strategies are known as pre-partitioning and post-partitioning. In post-partitioning, the program is designed and written as if for a single processor. After the program is completed, the desired partitioning is specified using separate software tools or is accomplished automatically by the distributed operating system (Hutcheon and Wellings, 1987:72). Pre-partitioning begins at the very start of the design process. A particular construct of the programming language is selected as the basis of parallelization and used to encapsulate each of the system parts that will run in parallel (Hutcheon and Wellings, 1987:72).

Several advantages are claimed for the post-partitioning strategy. One is that it promotes portable software by making it possible for the same program to be mapped onto different hardware configurations. Another is that, because the program is written as if it were to run sequentially, there are no restrictions on how the language is used. Finally, there is concern that the Ada language does not contain facilities for specifying the configuration of the software over the underlying hardware (Hutcheon and Wellings, 1987:72).

Automated post-partitioning, where the distributed operating system partitions the sequential program without intervention from the designer, would be the ideal situation. However, such an operating system would have to be extremely sophisticated to achieve efficient load-balancing, and the resulting overhead might make it unsuitable for real-time systems. In addition, automated partitioning could not take full advantage of the parallelism inherent in certain problems as Fox et al explain, "algorithms are often very different in their sequential and concurrent forms, and compilers cannot be expected to deal with such basic restructuring" (Fox et al, 1988:27). Real-time systems would require even more sophistication due to the critical timing constraints on portions of their software. The operating system would have to be aware of these when it made its load-balancing

decisions. In any case, no such operating system currently exists nor is one likely to be developed in the near future (Chow and Feridun, 1986).

Manual post-partitioning is possible using tools such as Honeywell's Ada Program Partitioning Language. This language allows the mapping details to be expressed in a specification completely separate from the program (Cornhill, 1984b). However, the feasibility of the tool remains to be demonstrated. In addition, the writing of a separate specification entails additional workload, which may not be justified. Important design decisions such as partitioning should be made early in the design phase when their effect on system operation can be estimated and included in the design. This is important because potential distribution may affect system interfaces (Hutcheon and Wellings, 1987:74).

Correct use of the Ada language in the pre-partitioning strategy can provide all of the advantages claimed for post-partitioning. Operations that lead to hardware dependency in a parallel system, such as task synchronization and dispatching, storage management, and exception handling, are handled in Ada by a machine-specific run-time environment and not compiler-generated code (Baker, 1985:Ch4,4). As a result, concurrent software written in Ada is not hardware dependent because the hardware dependent operations are left to the system's run-time environment.

Concern about restrictions on language use in this strategy are also unfounded. Although Ada code must be encapsulated in the task construct to run concurrently, the uniformity of the language reduces the impact this has on the programmer's productivity. For example, task entry calls mimic procedure calls and can be used in the same manner procedure calls would be used in a sequential program. With the task, all operations are available that would be available in a corresponding sequential program. Therefore, the use of the task construct places no unreasonable restrictions on the programmer.

The lack of configuration facilities in Ada can be overcome through the use of data driven design. Rather than depending on a separate cyclic executive, which would require configuration facilities to schedule and control independent tasks, data driven design depends on the run-time environment scheduler to control access to processors, based on which tasks are ready to run and their relative priorities. Because the Ada rendezvous

mechanism blocks tasks that are awaiting data, the flow of data through the system determines which tasks are ready to run (SofTech, 1986:Ch2,20). By controlling this flow of data, the designer can control the scheduling of tasks without extra-language configuration management facilities.

Pre-partitioning is the superior strategy for parallel design in Ada because it promotes early examination of critical design issues and eliminates the overhead of separate partitioning specifications. In addition careful design can yield all of the benefits of the post-partitioning strategy.

3.2.2 How to Partition. Given the pre-partitioning strategy and the Ada language, (Cornhill, 1984a) suggests four language levels at which partitioning can occur. Two of these, partitioning on any source program construct and extending Ada, require facilities outside of those provided by the Ada language. Because these facilities are currently unavailable or untested, these two methods will not be considered here. Of the remaining two methods, the first consists of writing a separate program for each processor. This method was commonly used in the past because of a lack of language constructs designed specifically for parallel processing. It is inefficient and has many disadvantages. First, this approach ties the software closely to the underlying hardware. If the hardware is later changed, the software must often be redesigned and rewritten. Any attempt at reallocating functions among processors will also require redesign (Cornhill, 1984b:364).

Another problem is that much of the reliability gained from the use of high order languages is lost. In Ada, semantic rules are enforced by the compiler only within a program and not across program boundaries (Cornhill, 1984b:364). Compilers may even generate different internal structures for logically identical data types in each of the programs leading to hidden problems when data is exchanged between them. Finally, communications capabilities in the Ada language are strong for passing data within programs, but the only facility available for communication between programs is inefficient and difficult I/O transfers (Cornhill, 1984a:155).

The other possible alternative is to partition on task boundaries by encapsulating the code representing each problem fragment within a task. The entire system is contained

within a single program with the task acting as the basis for concurrency. This method increases software portability, and reallocation of functions within the program can be handled by the run-time environment or with only localized changes to the affected task. The fact that the whole system is contained within a program allows for the full range of semantic checking and makes Ada's extensive communication capabilities available to the programmer.

Hutcheon and Wellings suggest that for a language construct, such as the task, to be effective as a partitionable boundary it must support: separate compilation, exception handling, and dynamic instantiation without reinitializing the entire system (Hutcheon and Wellings, 1987:72). Tasks support dynamic instantiation and exception handling. Although tasks are not separately compilable, separate compilation can be supported by encapsulating each task within its own package.

Partitioning on task boundaries is superior to writing separate programs for each processor because of software portability, increased reliability, and better communication capabilities. All the requirements for a boundary construct can be met by encapsulating tasks in Ada packages. Therefore, this research will be based on the pre-partitioning strategy using tasks encapsulated in packages as the concurrent process boundaries. The next section explains how tasks are used to abstract problem fragments.

3.3 The Ada Tasking Paradigm

The existence of the Ada task construct is the language's primary advantage in the development of parallel systems. "Ada is the first widely used programming language to provide concurrent programming facilities" (Burger and Nielsen, 1987:49). Most other high-order languages require the programmer to go outside the language and use host operating system facilities or specially written assembly language routines to meet the requirement of independent operation, synchronization, and inter-process communication. This increases the complexity of the programming task and decreases the portability and maintainability of the resulting code (Booch, 1983:232).

The use of Ada eliminates the need to go outside the language. The task itself is

the unit of independent operation, and the rendezvous provides for synchronization and communication. However, "these facilities are quite unlike the services provided by a typical run-time executive or operating system" (Burger and Nielsen, 1987:1-49). The differences have an important effect on the design and are the subject of this section. The first part is an overview of the task construct and its use. The last three parts describe the rules of tasking in the context of a task's life cycle.

3.3.1 The Concept of Tasks. Ada tasks are specified and created as data objects with the same scope and life-time rules as any other type of data object. Two differences between tasks and other data objects are that tasks may contain local data objects, and they are executable program units (Baker, 1985:2). The specification of tasks as data objects has several advantages. The first is the increased ease of programming made possible by the consistency with the data model. The programmer who is familiar with data types and objects does not have to learn a new specification system. Also, the use of task types allows for multiple declarations of identical tasks without having to write copies of the same code multiple times. Task types serve as templates for task objects in the same way that data types serve as templates for data objects (Cherry, 1984:58). Finally, the number of tasks in existence can be managed at run-time by dynamically creating and removing task objects using the allocator.

Each task follows an independent thread of control and is capable of operating asynchronously throughout most of its lifetime. Exceptions to this rule occur when it communicates with another task (a rendezvous), at the start and end of activation (when it must be synchronized with its parent), and at termination (Baker, 1985:3).

Tasks exist in one of four states: running, ready, blocked, or terminated. Running tasks are currently assigned to processors. Ready tasks are awaiting assignment to a processor. Blocked tasks are delayed or awaiting rendezvous. Terminated tasks are completed and require no further processing (Booch, 1983:237). The scheduling of ready tasks is handled by the run-time system. The programmer has some control over the method of scheduling through the use of static priorities which may be assigned to tasks in the coding stage (Baker, 1985:10).

The Ada task is an abstract model of concurrency that does not depend on the details of the underlying hardware. Ada tasks are always logically concurrent even if there are insufficient underlying physical processors for each task to operate simultaneously. If the number of tasks is greater than the number of processors, logical concurrency is created by the scheduler interleaving execution of the workload over the available physical resources. Logical concurrency is a distinct advantage to the designer who is free to abstract all the concurrency in the problem without knowing how many processors will eventually be available. It also increases the portability of concurrent programs from one size parallel system to another (Cherry, 1984:48).

The life cycle of the task can be divided into three parts: creation, operation and communication with other tasks, and completion. These three stages of the life cycle are the subject of the next three subsections.

3.3.2 Task Creation. Task creation consists of two parts: elaboration and activation. Elaboration takes place on the task specification and body and is the equivalent of the elaboration of a data object. In the case of a task, it establishes an entity that can be used to define execution of a task from that point on (Cherry, 1984:30).

The activation of a task occurs on the declarative part of the task body including any local data objects and the elaboration and activation of any local tasks (Baker, 1985:11). After activation, the task immediately begins processing its sequence of statements.

There is an important distinction between the invoking of subprograms and the creation of tasks. "The executions of task bodies are invoked automatically; but the execution of the procedure body must be invoked by an explicit procedure-call-statement" (Cherry, 1984:37). As a result, "the processor completes activation of the master's dependent tasks before it executes the first statement following the master's declarative part" (Cherry, 1984:36). These timing relationships must be carefully considered by the designer. A task can be made to initiate like a subprogram by making the first statement in the task an *accept* block. This causes the task to wait until the corresponding *entry* is called before beginning processing.

Tasks are not separately compilable and must be encapsulated in a subprogram or

package. Blocks and other tasks can also be parents of tasks. Tasks can be created in one of two ways: statically or dynamically. Static task creation occurs during elaboration of the parent's declarative part through a task declaration or an object declaration of a task type. This happens during importation of packages, invocation of subprograms, upon entering a block, or during activation of the parent task. Tasks are created dynamically through the use of an allocator for a task type. Dynamic allocation provides additional flexibility for task management during run time.

3.3.3 Task Synchronization (The Rendezvous). Once a task is activated, it begins to run automatically. If it operates completely independently, without need for communication with any other process, a task can complete its assigned processing and terminate without further hindrance. If, however, the task needs to communicate with another process, the Ada language provides the rendezvous for that purpose.

The rendezvous synchronizes two tasks, temporarily suspending their independent threads of control, in order to pass information between them (Nielsen and Shumate, 1988:21-22). The communication process begins when a calling task calls another task's *entry*. *Entry* declarations specify points at which tasks may be called and are similar to procedure specifications. Each *entry* in a task is matched by an *accept* block in the task body. This block contains the instructions that will be executed each time a call is accepted. This block is roughly the equivalent of a procedure (Cherry, 1984:85).

Once a calling task calls an *entry*, it is suspended from operation until that call can be accepted by the called task. When the call is accepted, any parameters are transferred and the *accept* block is executed. Upon completion of the block, the two tasks are released to resume their separate threads of control. If multiple calls are made to the same *entry*, the calls are queued at the entry and serviced, one at a time, on a first-come first-served basis each time the called task reaches the *accept* block (Nielsen and Shumate, 1988:22).

On the other hand, if a task reaches an *accept* block without being called, it is suspended until an *entry* call is received that matches that block. Although the rendezvous is a convenient form of synchronization, the designer must be careful of the possible occurrence of deadlock caused by a task waiting on an *entry* that is never called or calling an

entry that is never answered.

The asymmetry of the Ada rendezvous also affects design decisions. The calling task knows the name of the task it calls. The called task, however, does not know who is calling it. In addition, a calling task may only wait on a single entry queue while a called task may have multiple callers waiting in several accept queues. These factors must be considered by the designer when deciding which tasks will be active (call other tasks) and which will be passive (provide service to other tasks) (Nielsen and Shumate, 1988:22-23).

3.3.4 Task Completion. The final phase of the task cycle is termination. Termination is the third synchronization point in the life of a task. The effect of this last synchronization is determined by the concept of task dependency. A task is always dependent on a master construct which can be either another task, a block, a subprogram, or a library package. Tasks created by an allocator depend on the master that elaborates the task's access type definition while other tasks depend on the master whose execution created the task object. The master may not terminate before all of the tasks that depend on it have terminated (Flynn et al, 1987:55-56).

A task can terminate in one of three ways. If it has no dependent tasks, it completes after executing the last statement in its sequence of statements. If there are dependent tasks, the task may complete after the execution of its last statement and the termination of all of its dependent tasks. A third method of termination is the use of the *terminate* clause in a *select* statement. If a task reaches an open *terminate* statement, its master has reached the end of its sequence of statements, and every other task that depends on that master is completed or waiting at a *terminate* statement, then all tasks that depend on that master will be terminated together in a termination wave (Flynn et al, 1987:57).

Task termination rules add additional considerations to the design. The designer must ensure that all dependent tasks are properly terminated, or a master unit may become deadlocked. Also, if the order of termination is important, it must be controlled through careful selection of master/task dependencies and/or judicious use of the terminate statement.

3.3.5 Summary. The Ada task construct is ideal for abstracting concurrent operations into independent processes. It eliminates the need for operating system calls or assembly language routines to control parallel operations. Tasks operate asynchronously except during activation, rendezvous, or termination. This independent operation ensures parallel programming efficiency, but the resulting non-deterministic nature of the tasks must be carefully considered by the designer.

During activation, tasks can be kept from beginning too soon by making an *accept* the first statement in the task. This will effectively block the task from continued operation until it receives an *entry* call from a designated controller task.

Synchronization during operation is accomplished with the rendezvous. Its use should be kept to a minimum because of the associated run-time overhead, and because synchronous operation inherently limits the efficiency of parallel programs. Use of the rendezvous also raises the possibility of deadlock if *entry* calls go unanswered or tasks wait on *accept* statements whose entries are never called. The next section explains how well-known principles of software engineering can be applied in the multitasking environment to reduce the cost of synchronization and avoid deadlock.

3.4 Multitasking Design Issues

The application of structured design principles in the detailed design phase of a sequential implementation has been shown to lead to greater ease in system development and result in more modifiable, maintainable code (Yourdon, 1979). These principles can produce the same results in concurrent design if they are expanded to include multitasking concepts. This section begins by examining how the most important of these principles, cohesion and coupling, can be applied in the concurrent domain. It continues with an explanation of how intermediary tasks can be used to enforce loose coupling. Finally, the overhead of the tasking model is outlined to show its inherent computational costs.

3.4.1 Concurrent Coupling and Cohesion. Cohesion describes the strength of association between elements within a module, while coupling describes the strength of association between different modules (Yourdon, 1979:93). It is generally accepted that

strong cohesion and loose coupling leads to more structured design, easier modification, and higher maintainability. Although the task does not meet the strictest definition of a module because it cannot be called by another module and is not separately compilable, these restrictions can be relaxed to allow the application of the concepts of cohesion and coupling in a concurrent environment (Nielsen and Shumate, 1988:137-139).

The idea of cohesion can be applied to tasks in the same manner as more standard modules, but the coupling concept must be adjusted to account for the special interdependencies that result from concurrent operations. In this context, coupling can be considered at two levels: between subprograms and tasks and strictly between tasks. In the former case, coupling can be evaluated in the standard way as if between any two modules (Nielsen and Shumate, 1988:139). The latter case requires additional definition.

The differences in the concept of coupling during concurrent operations manifest themselves in the case of task to task interaction. Nielsen and Shumate call this "concurrency coupling" (Nielsen and Shumate, 1988:140). Tasks are considered tightly coupled if one calls the other's *entry* directly. There are various degrees of tightly coupled tasks. The tightest occurs during a rendezvous where *in out* or both *in* and *out* parameters are included in the call. In this case, the calling task requires a reply and the two tasks must remain synchronized for the entire period of data transformation. A lessor amount of coupling occurs when only *in* or *out* parameters are in the call. Here, the two tasks are only synchronized long enough for data to be copied from one to the other. A parameterless call represents the least amount of coupling (Nielsen and Shumate, 1988:140-141).

Tight coupling should be avoided whenever possible because periods of synchronization eliminate independent operation and thus reduce the efficiency of parallel processing. Loose coupling is achieved through the use of intermediary tasks between the caller and called task pair. These tasks perform a buffering function to ensure that the two main tasks can continue processing unhindered by unnecessary synchronization time (Nielsen and Shumate, 1988:142).

3.4.2 The Use of Intermediary Tasks. Nielsen and Shumate identify three varieties of intermediary tasks. A buffer task is a server only. It contains an entry to accept data

from a producer and an entry to provide data to a consumer upon request. In between producer and consumer calls, the data is stored internal to the task, usually in a queue. A transporter is strictly an active task. It requests data via an *entry* call to the producer task and then outputs the data via an *entry* call to a consumer. A relay task is a combination of the two. It waits until called by the producer and then immediately calls the consumer (Nielsen and Shumate, 1988:142-143).

Intermediary tasks are used in various combinations to achieve the desired degree of coupling between producer and consumer tasks. They are also useful to control caller/called relationships. For example, two tasks that operate at different speeds can be effectively uncoupled using a buffer task. The producer calls the buffer as soon as data is ready to output and then resumes processing. The data is stored in the buffer task until the consumer is ready to accept it. The efficiency of the parallel algorithm is reduced only by the time necessary for each task to synchronize with the buffer. A further increase in efficiency is possible by introducing transporter tasks on either side of the buffer. The transporter tasks are always waiting to communicate with producer and consumer and absorb the overhead of communicating directly with the buffer (Nielsen and Shumate, 1988:155).

Other combinations of intermediary tasks are possible and should be tailored directly to specific design requirements. A good concurrent design should have a balanced use of intermediary tasks with no cyclic dependencies and a minimum amount of busy waiting. It should also minimize the amount of processing done during a rendezvous (statements within *accept* blocks) and ensure appropriate modes are used for *entry* parameters (Nielsen and Shumate, 1988:140). The use of intermediary tasks is an excellent tool, but it does not come without costs. Task generation and rendezvous is a source of significant overhead and must be considered in the design.

3.4.3 Tasking Overhead. The need to reduce task coupling must be balanced against the resulting overhead of the intermediary tasks. "This overhead includes task activation and termination, task scheduling and dispatching, context switching, propagation of exceptions, selection of an open entry in a selective wait statement, queueing management

of entry calls, and allocation of task control blocks" (Nielsen and Shumate, 1988:156).

The amount of overhead associated with each of these operations is dependent on the specific run-time system implementation, but it will always be significant. One study done by Burger and Nielsen was done using DEC Ada on a Vax 8600. All results were normalized to the overhead of a procedure call as a representative baseline of what an equivalent sequential program would experience. The simple act of task activation and termination was found to cause 178 times the overhead of a procedure call, while a producer-transporter-buffer-transporter-consumer data transfer such as that previously described required 204 times the overhead (Burger and Nielsen, 1987:56). Even a single rendezvous is computationally intensive. A study by Baker shows that a parameterless entry call generates 15 times the overhead of an equivalent procedure call (Baker, 1985:6).

These studies were done using single processor implementations, and some of the overhead can be attributed to context switching. Multiprocessor implementations will experience less of this type of overhead but will have to contend with interprocessor communication time. A shared memory machine has an advantage in this area because its interprocessor communication time is nominal. Therefore, an efficient shared memory implementation should result in lower overall overhead compared to these figures.

The overhead attributable to each of these activities will vary depending on the hardware architecture and the particular run-time implementation. The designer should take careful note of the extent of this overhead and consider it carefully in all of his parallel design decisions. The next section explains how the use of Ada tasks supports a new concept of process control called data driven design. This method allows for superior scheduling efficiency and minimizes overhead.

3.5 Data Driven Design

In the past, most real-time systems have been developed using the concept of a cyclic executive. The cyclic executive runs as a master process, directly under the control of the operating system, and handles the scheduling of all other processes. Scheduling is done in a deterministic fashion based on a task list preset by the programmer in the initial

coding of the system (SofTech, 1986:Ch2,1). Before it is installed, the preset schedule is optimized by the programmer to meet timing and processing requirements and to match the particular hardware configuration at hand. This method of handling real-time control is very inflexible due to the hardcoded nature of the task list. It has little ability to adapt in order to take advantage of processor idle time, hardware upgrades, or changes in time or processing constraints.

The use of a cyclic executive was necessary in the past because commonly used programming languages had no inherent concurrent control features. The Ada language makes an alternate real-time control concept possible. It is called data driven design (SofTech, 1986:Ch2,20). In Ada, scheduling of tasks is accomplished by the run-time system scheduler, based on which tasks are in ready status and their relative priorities. Whether a task is ready or not depends on whether it is blocked for rendezvous or has all the data it needs to continue processing. Scheduling control is applied to the system by the flow of data. Additional control for high-priority tasks is available through the use of the Ada priority pragma.

Data driven design results in minimum processor idle time because ready tasks do not have to wait for the start of their frame time as in the case of the cyclic executive. Instead, they are scheduled as soon as a processor is available. In addition, data driven designs adapt automatically to changes in hardware resources, timing or processing requirements because scheduling decisions are made in real-time by the run-time system based on the availability of data and processing resources.

The data driven design is sensitive to changes in the data flow. Here, automatic adaptation is impossible. However, if the designer has carefully modularized data entry points at each task and followed the principle of loose concurrent coupling, all that is necessary is a change in the entry points and intermediary tasks to reflect the new flow of data.

This method also eliminates the need for time frames and, therefore, the possibility of frame overrun. On the other hand, timing problems are still possible if processing requirements outpace available processing resources. Because high priority tasks are always

scheduled first, any timing problems will show up in low priority tasks first (SofTech, 1986:Ch3,8). The result is a throughput problem in the overall system. Since the processor idle time is already at a minimum, the only way to solve this problem is by increasing processor resources or decreasing computational demand.

Given the correct design methodology, the concepts of data driven design can be built directly into the applications software. This eliminates the need for a cyclic executive and reduces the workload on the programming team. A suitable design method is described in the next section.

3.6 Software Design Methodology

Having completed the analysis portion of the life cycle, a comprehensive software design method is now necessary to complete the abstraction from the problem space to a design implementable on a digital computer. This method must be compatible with each of the decisions made in the earlier phases of the project. In this case, the problem is a real-time system. This places several requirements on the design method. "Real-time designs typically include a set of concurrent processes that operate asynchronously to accommodate the different speeds of the various hardware devices" (Nielson and Shumate, 1987:695). "The processes usually have to communicate, and synchronization points are included in the designs to provide for message passing and protection of shared data" (Nielson and Shumate, 1987:695). "The software design must also identify algorithms, data structures, and control structures to convert input data to internal data representations and output data" (McGee, 1987:5). There must be a method of identifying units of compilable code and designing error detection and recovery (McGee, 1987:5).

The selection of Ada as the software programming language places its own demands on the design. "Ada requires a new design method for real-time systems since it differs from conventional language/executive combinations in two important ways: (1) concurrency and process communication are inherent in the language and do not require a separate real-time executive; and (2) the nature of the tasking model is different from current popular approaches for task scheduling, communication, and synchronization" (Nielson

and Shumate, 1987:696). Finally, the methodology must support software pre-partitioning at the task and package level.

Most commonly used design methods fall short in one or more of these areas. One that was specifically developed for designing large, real-time distributed systems in Ada is the Layered Virtual Machine/Object-Oriented Design (LVM/OOD) method. It provides capabilities for describing concurrent processes and the communication between them. It also supports real-time considerations within the same design tools. Its basis for concurrent processes is the Ada task, groups of which are encapsulated into packages. The rest of this section is an overview of LVM/OOD abstracted from (Nielson and Shumate, 1987) and (Nielson and Shumate, 1988). The following is meant only as an overview and the above sources should be consulted for more specific information.

3.6.1 Overview. Good software design requires the successful integration of two components: algorithms and data structures. Both of these components are of equal importance, but they often raise differing concerns. LVM/OOD deals with these concerns by combining two design concepts: layered virtual machines and object oriented design.

Layered virtual machines are abstractions of algorithms into independent processes capable of operating in parallel. These processes are virtual because they are not associated with underlying hardware. The run-time environment is responsible for the binding of processes to processors. The use of layering creates a hierarchical set of modules that defer implementation details and support information hiding.

Object oriented design balances the usual emphasis placed on functional design. OOD is used to abstract problem space objects into software data structures and their associated operations. Object abstractions can be either object managers which encapsulate data structures or type managers which encapsulate definitions that form templates for the creation of data structures. These managers provide the means for describing data and are especially important in data driven designs.

3.6.2 System Design. The layered virtual machine and object oriented design concepts are combined into an eight step design methodology. The first four steps are high

level and comprise the system design. The first step is to determine the hardware interfaces to the control system. These are illustrated using a context diagram. Each of the hardware devices is depicted separately while the internal control system is represented in abstract form as a single entity. High-level inputs and outputs are labeled on the interfaces.

The second and third steps of the design are based on an edges-in development philosophy. First, each of the external devices, or edge functions, are assigned separate control processes. These are simple device drivers and contain the bare minimum of instructions. Once all of the edges are described, the middle part (controller) is decomposed into its primary components. Data flow diagrams (DFDs) are used to show the interaction among these components. Very complex components can be further decomposed using hierarchical levels of DFDs.

The final step in the system design phase is to determine concurrency among the controller components. The goal of this step is to abstract these components into processes that will run independently with a minimum of inter-process communication and dependence. A successful abstraction will yield the best balance between concurrent operation and the overhead inherent in process control.

Concurrency considerations will often lead to a grouping of several components into a single process. Nielsen and Shumate provide several rules to assist in this task. The concept of functional cohesion suggests that components with closely related functions can be grouped into a single process to reduce overhead. In a like manner, temporal cohesion collects operations that occur during the same time period or after the same event. In both cases, if the functions involved are small, sequential, or closely dependent, significant overhead can be saved without appreciable loss of performance.

There are also some functions that should be left as separate processes. Time-critical components should be implemented as separate processes with appropriate priorities to ensure that real-time constraints are met. Periodic functions should not be combined with operations that run in differing periods. Finally, background processes should also be separate to preclude interference with time-sensitive ones and to provide for the best use of excess processor time.

This step focuses only on the concurrency of the high-level components indentified in the system DFDs. Additional partitioning may be possible at a lower level. Nielsen and Shumate recommend against having more than one level of concurrent tasks, but ignoring lower level partitioning may forgo performance gains. Therefore, this thesis will examine further partitioning in the final step of the detailed design.

When all of the components identified in the system DFDs have been abstracted into processes, the system design is complete. This portion of the methodology is ideal for real-time distributed systems because it provides methods for describing hardware interfaces, timing constraints, and concurrent processes. Up to this point, the design is independent of the implementation language. The next phase of the design is geared toward using the unique features of the Ada language.

3.6.3 Detailed Design. These last four steps of the methodology comprise the detailed design. The system design phase provides a high-level abstraction of the problem in the form of a set of communicating sequential processes. The purpose of this phase is to transform that representation into the full description of algorithms and objects necessary for an efficient Ada implementation.

The first step of this phase is to determine what type of interfaces exist between the processes created in the final phase of the system design. These interfaces define the communication between the processes and can be one of several forms: messages where a reply from the receiver is required, data transfers where no reply is required, signals used to coordinate on the occurrence of certain events, and shared data access. The type of communication determines the amount of coupling between processes. Messages that require replies have the highest coupling followed by simple data transfers. The least coupling is caused by event signals. Shared data access requires additional attention and must be protected by some intermediate task to provide for mutual exclusion. Once these interfaces are identified, processes and their corresponding interfaces are depicted using a process structure chart.

The concept of process coupling is important because it describes the degree of dependence between processes and, therefore, how free they are to operate in parallel.

The goal of a good concurrent design is to reduce coupling as much as possible. Step two of the detailed design seeks to do this by introducing intermediary processes into the design. First, the processes are translated into Ada tasks. Between tightly coupled tasks, additional tasks are added whose sole purpose is to facilitate communication and thereby decrease coupling. These intermediate tasks can be of three types: buffer tasks to store information received from a producer until a consumer calls for it, transporter tasks to request items from producers and transfer them to consumers, and relay tasks to wait for data from a producer and then call a consumer. Combinations of these task types are used to reduce task dependency. The result of this step is the Ada task graph.

The next step is to encapsulate the tasks into packages. At the same time, the objects used for communication between the tasks should be abstracted into data objects and encapsulated into packages as well. Although tasks can reside in other tasks and subprograms as well as packages, Nielsen and Shumate suggest that they be placed in packages to increase modularity, portability, and reusability. They also provide several rules for task encapsulation. Tasks should be grouped into the same package if they have similar functions or if their general nature makes them good candidates for reuse. Coupling between packages should be minimized with respect to data types, operators, and constants by localizing these items to the package or group of packages in which they are actually used. Finally, the package should minimize the visibility of task entries to that which is essential for package interfacing. The products of this step are an Ada package graph and the corresponding OOD diagrams.

The final step is the further decomposition of large tasks. In this stage, each of the major tasks is reduced to its components using the layered virtual machine method. This may result in another level of concurrency which can be depicted using process structure charts and Ada task graphs, or it may result in a sequential decomposition which is described with structure charts. The lower level objects must also be abstracted into local data objects. The results of this step are shown using OOD diagrams.

3.6.4 Summary. The LVM/OOD methodology meets all the requirements for use in this type of software environment. It provides good rule-based decision making support

at both the system and detailed design levels. In addition, graphical tools are available at each step to enhance understanding of the problem.

Each level of the problem abstraction is supported. The edges-in approach specifies interfaces to hardware devices, and the concept of intermediate tasks satisfies the need to account for differing device speeds as well as loose coupling in process communication. Design rules are available to identify processes that will operate in parallel and how they will communicate if necessary. The layered virtual machine concept specifies all algorithms while object oriented design does the same for data structures.

The method is ideal for the use of Ada. The Ada task is the basic unit of partitioning, and the Ada tasking paradigm is directly supported. Units of compilable code are divided into packages, and concepts of modularity and information hiding are strongly supported.

3.7 Chapter Summary

This chapter was meant as a guide for the design and analysis of any parallel software development implemented in Ada. The first step in the analysis phase is to decompose the problem into independent fragments using the decomposition methods of relaxation, pipelining, and partitioning. After the problem is decomposed, the problem fragments are mapped to a corresponding language construct.

In Ada, the superior method uses the task construct for the basis of all independent processes. The task is well-suited for this purpose, supporting separate compilation (when encapsulated in a package), exception handling, and dynamic instantiation. The task itself is the unit for independent operation and the rendezvous makes it possible for tasks to communicate.

The use of the task as the basic programming construct requires some reapplication of standard software engineering principles. In particular, the design must ensure that tasks are not tightly coupled, or the resulting implementation will lose much of the benefits of parallelization. Tight coupling can be overcome through the use of intermediary tasks, but the inherent overhead of the tasking model must always be considered when deciding whether to insert additional tasks into the design.

The use of tasks also enables the designer to take advantage of a new method of process control called data driven design. Data driven design eliminates the need for a cyclic executive. It also results in superior scheduling efficiency and minimizes overhead.

Once the analysis is complete, a comprehensive software design method is needed to finish abstracting the problem. The Layered Virtual Machine/Object Oriented Design method supports the special needs of real-time development and the Ada language while encompassing standard software engineering principles.

IV. Kalman Filter Design

The previous chapter presents a general set of guidelines for abstracting a large, real-time or simulation problem into a design suitable for implementation in Ada on a parallel computer system. The remainder of this thesis is devoted to an example of such an implementation: a Kalman filter tracking system. This problem includes both a real-time portion and a simulation portion and is therefore ideal for investigating the concepts discussed in the first three chapters.

The purpose of this chapter is to describe the Kalman filter tracking system design process from the initial problem analysis to the detailed design. The organization follows that of Chapter 3 and is intended to show how the concepts described there are applied to a real-world problem. The first section serves as a high-level analysis of the problem. Section 2 describes the decomposition of the Kalman filter system. Section 3 covers the system design, including: determination of the hardware interfaces, decomposition of the middle process, and identification of concurrent processes. Section 4 presents the detailed design including: determination of the process interfaces, introduction of the intermediary processes, and further decomposition of the large tasks. It also discusses how a control system based on the concept of data driven design was incorporated.

4.1 Problem Analysis

What follows is a specific analysis of the Kalman filter tracking algorithm chosen for implementation. It is based on research done at the Air Force Institute of Technology and on a sequential FORTRAN program that was developed to test the results of that research. This analysis is drawn from (Tobin, 1986), (Tobin and Maybeck, 1987) and (Norton, 1988), as well as a thorough review of the operational FORTRAN code. One important note is necessary here. The FORTRAN program is extensive, having been under continuous modification for over ten years, and it contains numerous simulation, test, and statistical routines, not all of which are necessary to run the simulation at any one time. This work is not a complete implementation of that program. Rather, it is a representative subset of those routines selected to keep the scope of this research manageable.

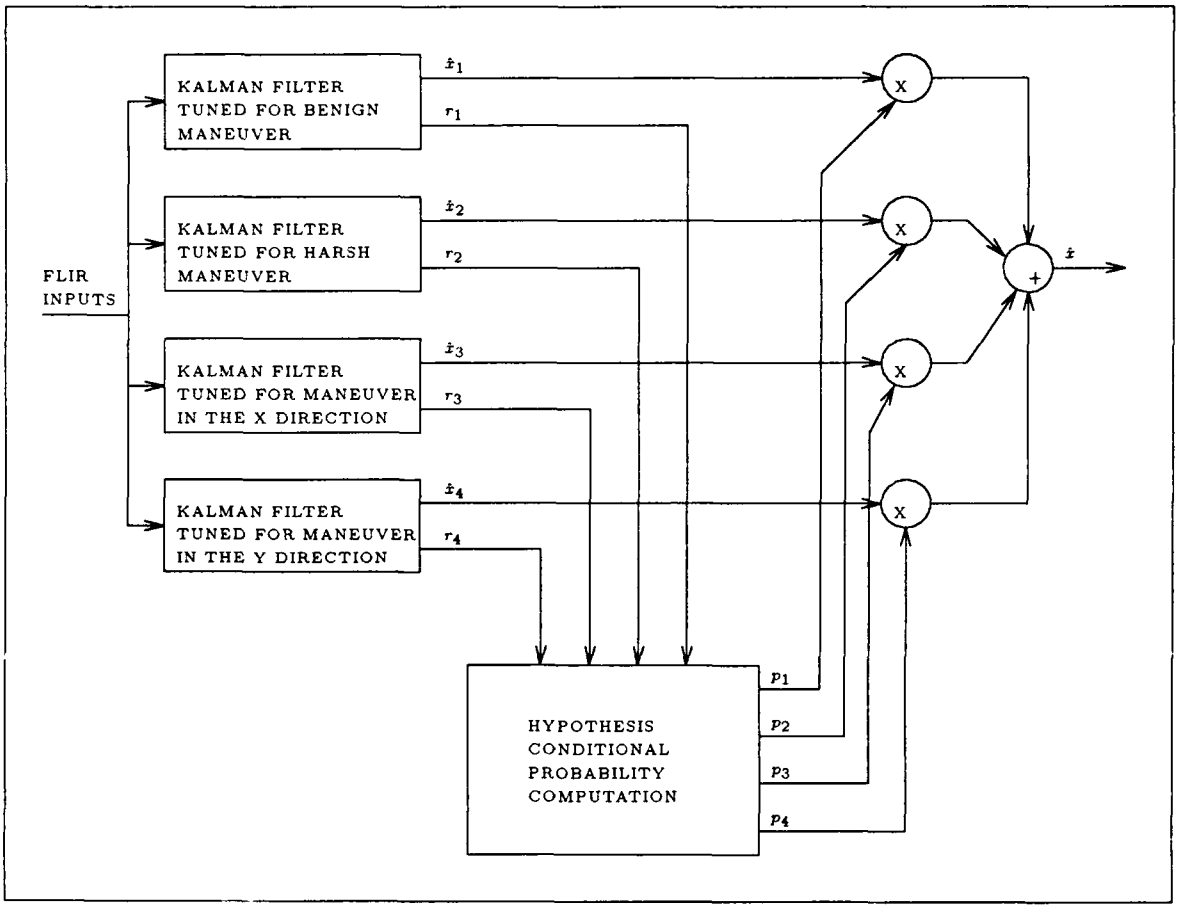

Figure 3. Multiple Model Adaptive Filter

The system selected for implementation consists of four separate Kalman filters that operate independently on the image data input from the forward looking infrared sensor. Each of these filters is tuned for a different type of target dynamics, ranging from relatively benign to highly maneuverable and including filters to accept jinking in either the azimuth or elevation direction of the FLIR image plane. The estimated state information from each filter, together with its residual, is fed into a conditional probability generator. The output from this generator is the degree of weighting each filter receives in the current frame. The probabilities are multiplied by their respective filter state outputs and the results from all filters are added to produce the propagated estimate to drive the laser pointing controller. A graphical description of the system is shown in Figure 3.

Raw FLIR image data is input into the system in the form of a single 8x8 array

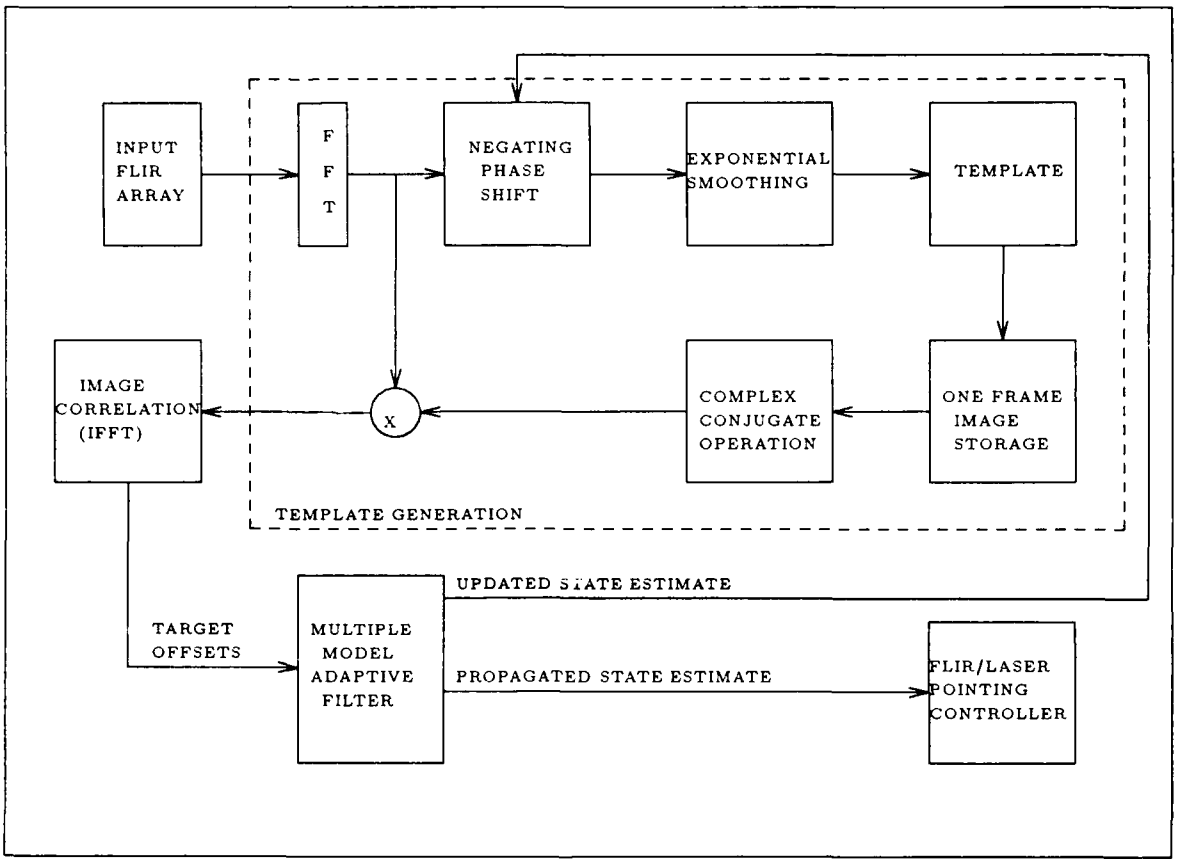

Figure 4. Diagram of Correlator/Kalman Filter Tracking System

of pixel intensity levels. This array represents the tracking window and is a subset of a larger image input from the FLIR at a rate of 30Hz. The relatively high input rate justifies the use of the relatively small tracking window. This data is fed into a single enhanced correlator. The resulting "pseudo-measurement" of the coordinates of the target center relative to the center of the FOV drives the bank of linear Kalman filters. A block diagram is shown in Figure 4. The block marked Multiple Model Adaptive Filter is the equivalent of Figure 3. From this point on, the Image Correlator together with all of the operations shown in the block marked Template Generation will be referred to as the correlator.

The purpose of the correlator is to produce a form of linear measurements from a non-linear image (the target image input from the FLIR sensor) in order that it may be processed by a linear Kalman filter bank. This effect is achieved by transforming the image data using a fast Fourier transform (FFT), multiplying the result with the complex

conjugate of the target template, and performing an inverse FFT. The result is a set of "measured" offsets of the target centroid from the center of the field of view which can be input directly into a linear Kalman filter, eliminating the need for more computationally intensive extended Kalman filters. The state information is processed within each of the four filters and the results are combined, using the Bayesian approach, in the manner discussed in Chapter 2 to yield an updated state estimate of the current target position and a propagated state estimate of its estimated position in the next frame.

In addition to providing the updated state estimates for the current frame, the tracking algorithm must also generate the next target template. This is done using the same transformed input image data from the FFT. A negating phase shift, using the updated state estimate from the filter, is used to reconstruct the transform of the target image centered in the field of view. The resulting image is exponentially smoothed with the previous template to approximate true finite-memory averaging. The output is the FFT of a template associated with a centered target image. This template is used to correlate with the input data in the following frame.

The foregoing explanation describes the tracking algorithm as it would be implemented on an embedded system. In order to test and adjust it without the benefit of sensors, a pointing system, and a real target, a rather complex simulator is necessary. Because this example will be implemented on a mainframe computer, the simulator will be included as part of the design. From this point on, the design of the tracking system and the simulator will continue in tandem. Necessary distinctions will be drawn between the two in the text and related figures.

The simulator consists of processes on both the input and output sides of the tracking system. The input side of the simulator is concerned with producing a stream of target images which mimic what a FLIR sensor would provide. This is done using two separate routines. The first is a track generator which uses a truth model to produce a representation of a target track mathematically. The second is a noise generator which produces random background noise to simulate physical phenomena that would interfere with the validity of FLIR sensor readings. The results from these two routines are combined by a third process into an array representing a simulated tracking field of view. On the output side,

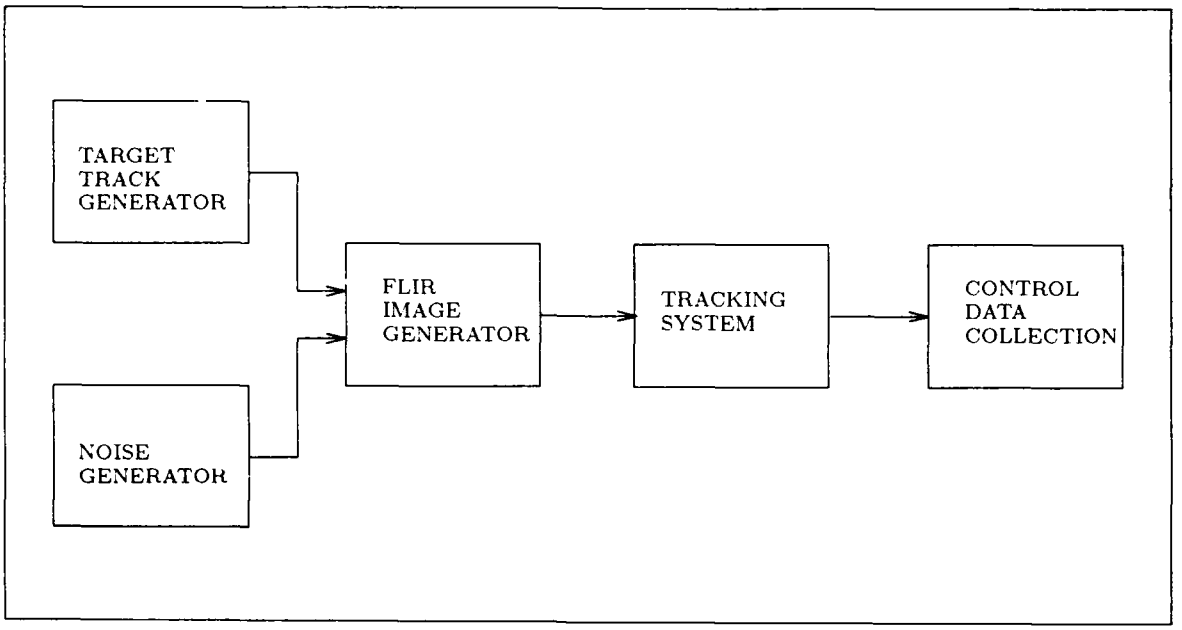

Figure 5. Kalman Filter Tracker with Simulator

the control information normally fed to the laser pointer must be captured so that it can be compared to validated results to determine the accuracy of the resulting system. The addition of the simulator is shown in Figure 5.

4.2 Decomposition of the Kalman Filter Tracking System

The previously described Kalman filter tracking system is multilayered, and the possibilities for problem decomposition can be explored on three different levels. The highest is the system level represented by Figure 4. At this level, a single correlator feeds a bank of Kalman filters. Within the correlator, several complex image processing operations are performed in sequence on a single data image array. Each of these functions requires significant computation, and, other than the data array, there is no communication between them. As a result, there is little data contention, and the workload is sufficient to outweigh the parallel overhead. This level meets the standards established in Chapter 3 for promising decomposition.

The dependence on a single data array and the fact that the image processing functions must be executed in sequence makes relaxation unsuitable for this level. In addition,

the operations are not homogeneous, which rules out partitioning as a viable option. However, if it is possible for each of the correlator image processing functions to process a different data array during the same frame, pipelining may be used to provide significant speed-up. In this case, the load could be balanced by assigning a separate function to each processor.

The middle level is the multiple model adaptive filter represented in Figure 3. At this level, there are four individual Kalman filters. Each of them operates independently to update and propagate system state information. In addition, the numerator term of the hypothesis conditional probability evaluation (see Equation (10)) can be computed separately for each filter and passed to the Bayesian averaging routine. The computational load for each filter is sufficient to warrant the resulting parallel overhead, and no communication is required between filters, eliminating the chance of data contention.

The filters operate in parallel, so pipelining would provide no benefit. Also, because each filter uses different parameters, the operations are not homogeneous and partitioning is not viable. The independence of data and the fact that no synchronization is required between filters makes this level ideal for relaxation. Using this method, load balancing can be achieved by assigning each filter to a separate processor.

The lowest level at which decomposition is possible in this system is in certain support routines. Both the correlator and the Kalman filters require matrix support routines including: multiplication, inversion, Fourier transform, and image phase shifting. In addition to these, the simulator also requires a Cholesky decomposition routine. These operations are homogeneous in nature and are frequently performed on data arrays which are large enough to justify parallel processing overhead.

These support routines can benefit from partitioning. By dividing the matrices up and processing each part on a separate processor, the computational load can be more fully balanced. However, because these operations will be performed on matrices of varying sizes, the routines must be adaptive to ensure an efficient match between the number of tasks spawned and the relative size of the matrix. Any static routine will lose the speed-up gained in processing large matrices due to the overhead and data contention generated in

smaller cases.

The simulation program also provides opportunities for decomposition. In addition to the support routines which it shares with the tracking system, relaxation can be used to parallelize the FLIR image generator. The noise generator, track generator, and image generator all operate asynchronously. There are no data dependencies other that the track and noise arrays passed from their respective generators to the image generator. Each of the functions is significant enough to warrant the resulting overhead and data contention is minimal.

In short, the Kalman filter tracking system presents numerous opportunities for decomposition. At the highest level, pipelining is possible in the correlator. Relaxation is also promising for use in the simulator. At the filter bank level, relaxation can be used to balance the load by assigning each filter to a separate processor. Finally, partitioning may result in the more efficient processing of many of the computationally intensive matrix operations. With the problem analysis and decomposition complete, the next step is the system design.

4.3 System Design

This section outlines the system design of the multiple model adaptive Kalman filter tracker using the LVM/OOD methodology. The graphs described by (Nielsen and Shumate, 1988) are used to portray the development of the design.

4.3.1 Determination of Hardware Interfaces. The particular tracking algorithm under consideration processes data inputs from a FLIR sensor. The data is in the form of a two-dimensional array of pixel intensities representing the hot spots of the target being tracked. The output from the tracking algorithm is the target's expected position in the next frame. This information is used to operate the laser platform pointing mechanism. This system has a single hardware input, the data array from the FLIR sensor, and a single hardware output, the control signals for the pointing mechanism. In this example, both the input and output devices are simulated in software. The resulting context diagram is shown in Figure 6.

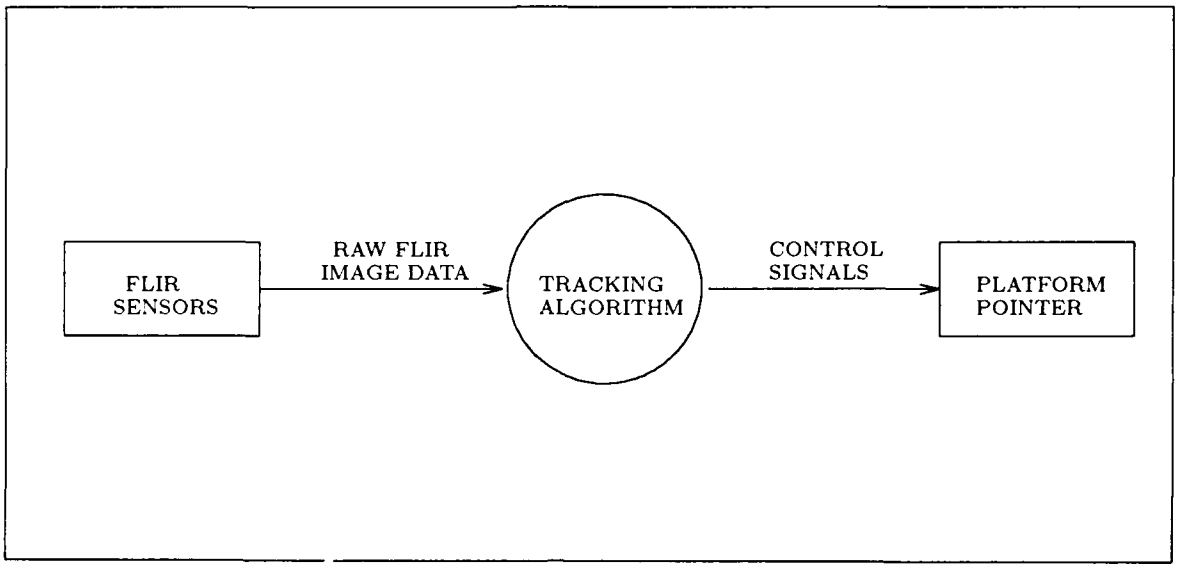

Figure 6. Context Diagram

4.3.2 Assignment of Processes to Edge Functions. Because there are only two hardware interfaces, there are only two edge functions. A single input task and a single output task are required. The FLIR input handler accepts data from the FLIR sensor in the form of a image array and passes it to the tracking system. The control signal output handler accepts the X and Y coordinates of the next expected target position from the tracking system and passes that information to the pointing mechanism. The preliminary concurrent process graph is shown in Figure 7. In the simulation environment, the FLIR input handler will accept a simulated FLIR image from an image generator implemented in software. The output handler will transfer control information to a collector process where it will be used to validate system accuracy.

4.3.3 Decomposition of the Middle Part. The tracking system consists of a single enhanced correlator which accepts the raw FLIR sensor input. After the input data is transformed into the Fourier domain, it is correlated with a predefined target image template created in the previous frame. The result of this correlation is a linear offset measurement given in terms of the X and Y coordinates of the target center in the image plane. This information is provided in parallel to each Kalman filter in the multiple model adaptive filter bank.

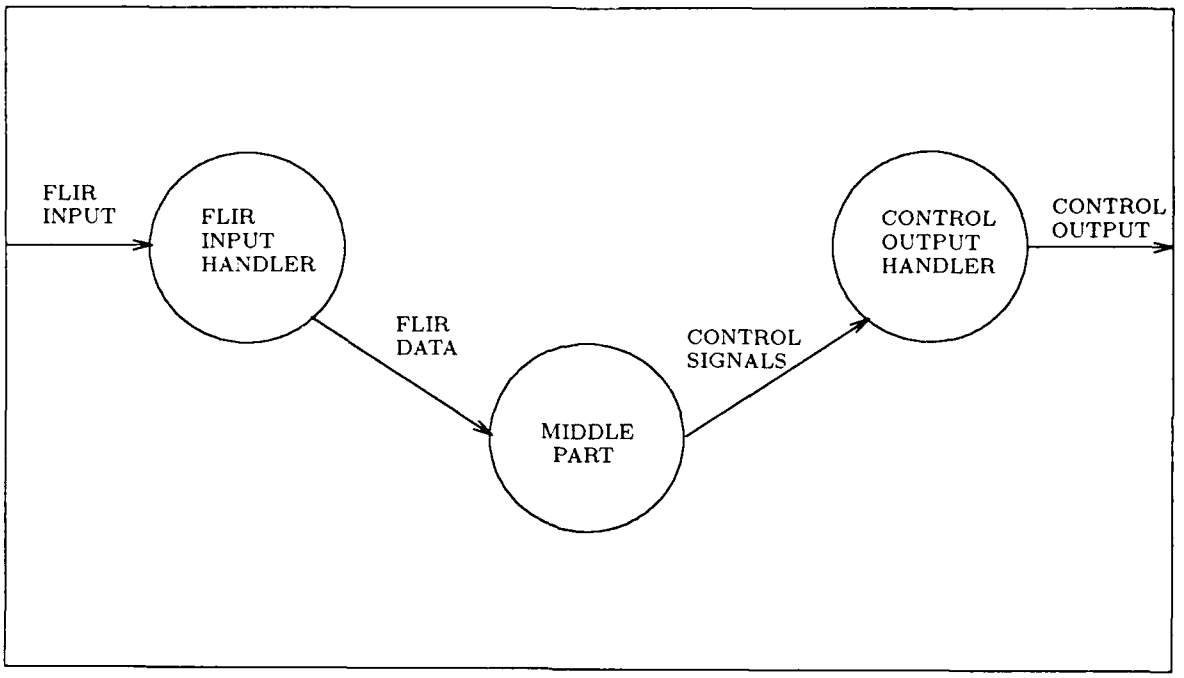

Figure 7. Preliminary Concurrent Process Graph

The same transformed input data is used to create the template for the next frame. The data is shifted in the Fourier domain using coordinates calculated from the updated state estimate generated by the Kalman filters. This shifted image is then exponentially smoothed with the previous template to become the new template.

The centroid information is used by the four Kalman filters to update the current target position estimate and propagate an estimated position of the target in the next frame. Because each filter is tuned for different target dynamics, the updates and estimates from each filter are different. These are combined based on each filter's probability of matching the true target dynamics. This probability is calculated using residual information provided by the filter update routine. The resulting combined update is returned to the correlator to produce the template for the next frame. The combined propagated position estimate is used to calculate the control inputs necessary to move the pointing device. Figure 8 is a data flow diagram of the complete Kalman filter tracking system.

In order to test the accuracy of the algorithm, it is necessary to simulate the FLIR sensor inputs. This is done by using a truth model process to generate the simulated track

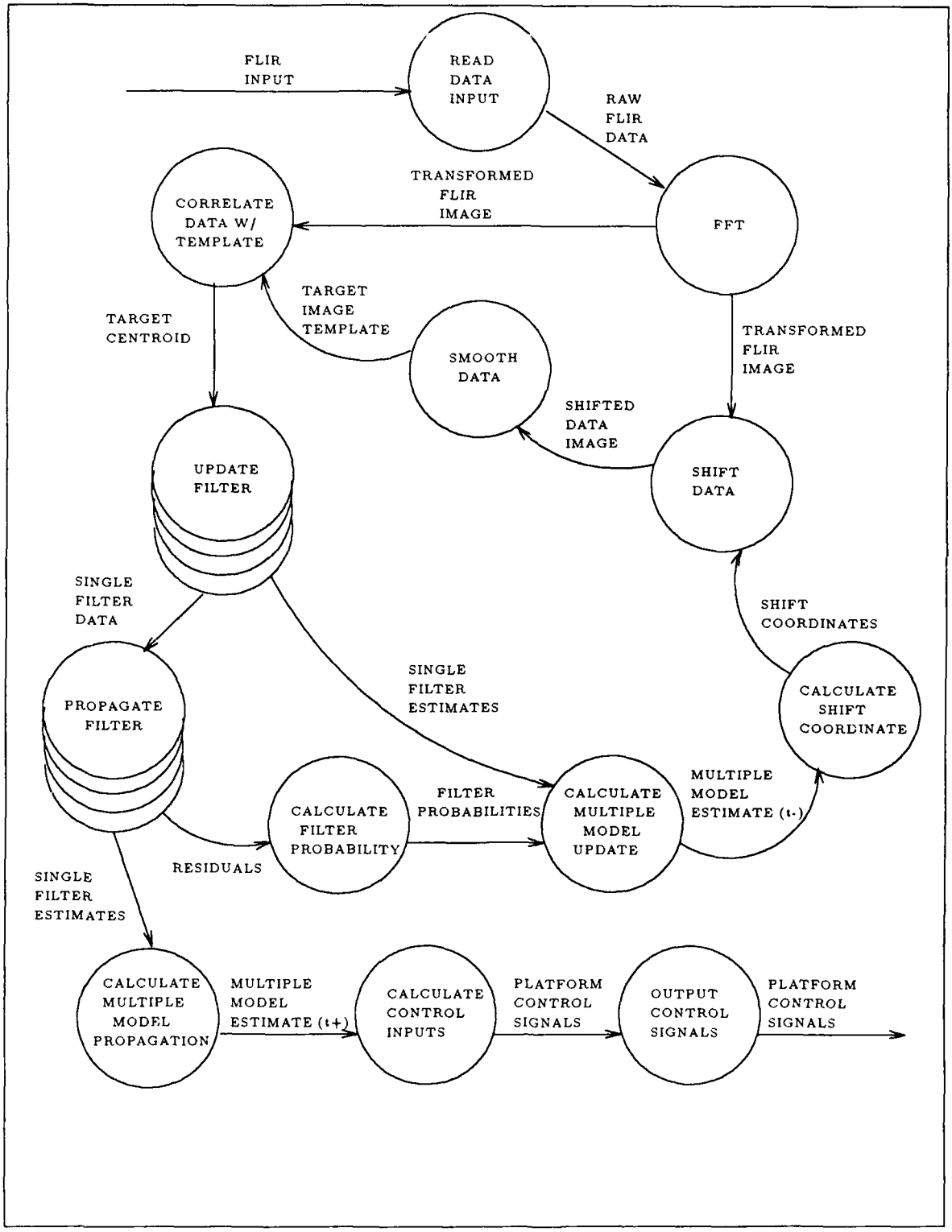

Figure 8. Tracking System Data Flow Diagram

of the target. Another process generates random noise which simulates sensor limitations, atmospheric distortions, and other real-world phenomena. The outputs of these two processes are combined to create a simulated FLIR input. The resulting data array becomes the FLIR sensor image and is passed to the tracking system input routine. It is also necessary to have a controller which initializes each of the simulation routines, the correlator, and the individual filters. This controller serves as the main routine in the implementation. Figure 9 shows the tracking system with the simulation processes in place.

4.3.4 Indentification of Concurrent Processes. Three possible levels of concurrency were discussed in Section 2: system, filter bank, and support routine. The parallelization of the support routines is not directly a part of the design and is discussed in Chapter 5. Because of the sequential nature of the correlator, the system level was a candidate for pipelining. Further study, however, proved pipelining to be impractical. The reason is the feedback loop within the correlator. This is evident in Figure 8. After completion of the initial FFT, the input data is immediately correlated with the target template. The creation of this template requires the data from the previous frame to pass through the entire processing cycle. It is therefore impossible to begin processing the data from the next frame until all processing on data from the previous frame is completed. This rules out pipelining and, consequently, any parallelization at the system level. As a result, the entire correlator is assigned to a single process. In addition, because the calculation of relative filter probabilities and the multiple model estimate must also occur in sequence (see Figure 8), these functions were also assigned to the same process as the correlator.

The input portion of the simulator provided better opportunities for parallelization. The track generator, noise generator, and image generator all operate independently with well-defined and infrequent interfaces. To take advantage of this, each of these is assigned to a separate process, as is the gaussian noise generator. The filter bank is also parallelizable. Each filter is assigned to a separate process and operates independently from the others and the correlator. Thus, relaxation proved to be a valid method of parallelization for both the simulator and the Kalman filter bank.

After completion of the system design, this example problem consists of eleven con-

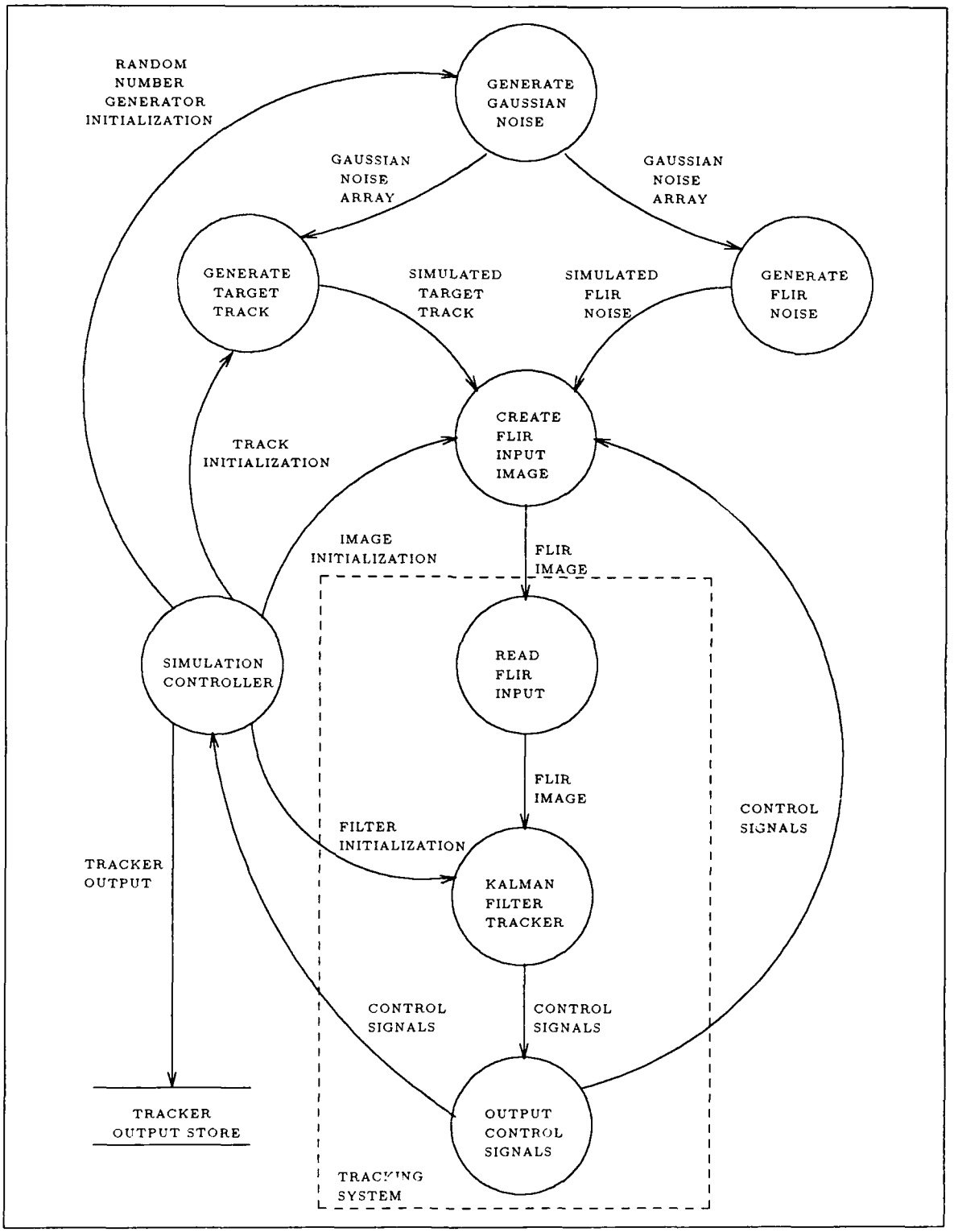

Figure 9. System Data Flow Diagram Including Simulator

current processes. Seven of these comprise the tracking system: the four Kalman filters, the correlator, and the FLIR input and control output handlers. The other four are associated with the simulator: the gaussian noise, FLIR noise, target track, and input generators. With these processes defined at an abstract level, it is now possible to begin the detailed design.

4.4 *Detailed Design*

This section outlines the development of the detailed design for the Kalman filter tracking system using the LVM/OOD methodology. Some changes and additions have been made. These are noted at appropriate points in the text. A discussion is included on how the concept of data driven design is incorporated to control task scheduling.

4.4.1 Determination of Process Interfaces. In the last phase of the system design, the various functions of the tracking system were decomposed into independent processes. The purpose of this first phase of the detailed design is to determine the degree of communication necessary between processes and the resulting level of task coupling. Figure 10 shows these relationships graphically.

The FLIR noise generator is highly coupled to the gaussian noise generator. Because both *in* and *out* parameters are exchanged, the FLIR noise generator must remain blocked during the entire time the random noise is being generated. On the other hand, the FLIR noise generator is only moderately coupled to the image generator. Once the simulated FLIR noise array is passed to the image generator, the two tasks may operate independently again. One additional advantage is that the FLIR noise generator outputs are not in any way dependent on previous outputs. Therefore, simulated FLIR noise can be generated asynchronously to the tracking system and buffered at the image generator.

The track generator is similar to the noise generator in all these respects. It too is highly coupled to the gaussian noise generator and moderately coupled to the image generator. Simulated target tracks may also be generated asynchronously and buffered at the image generator.

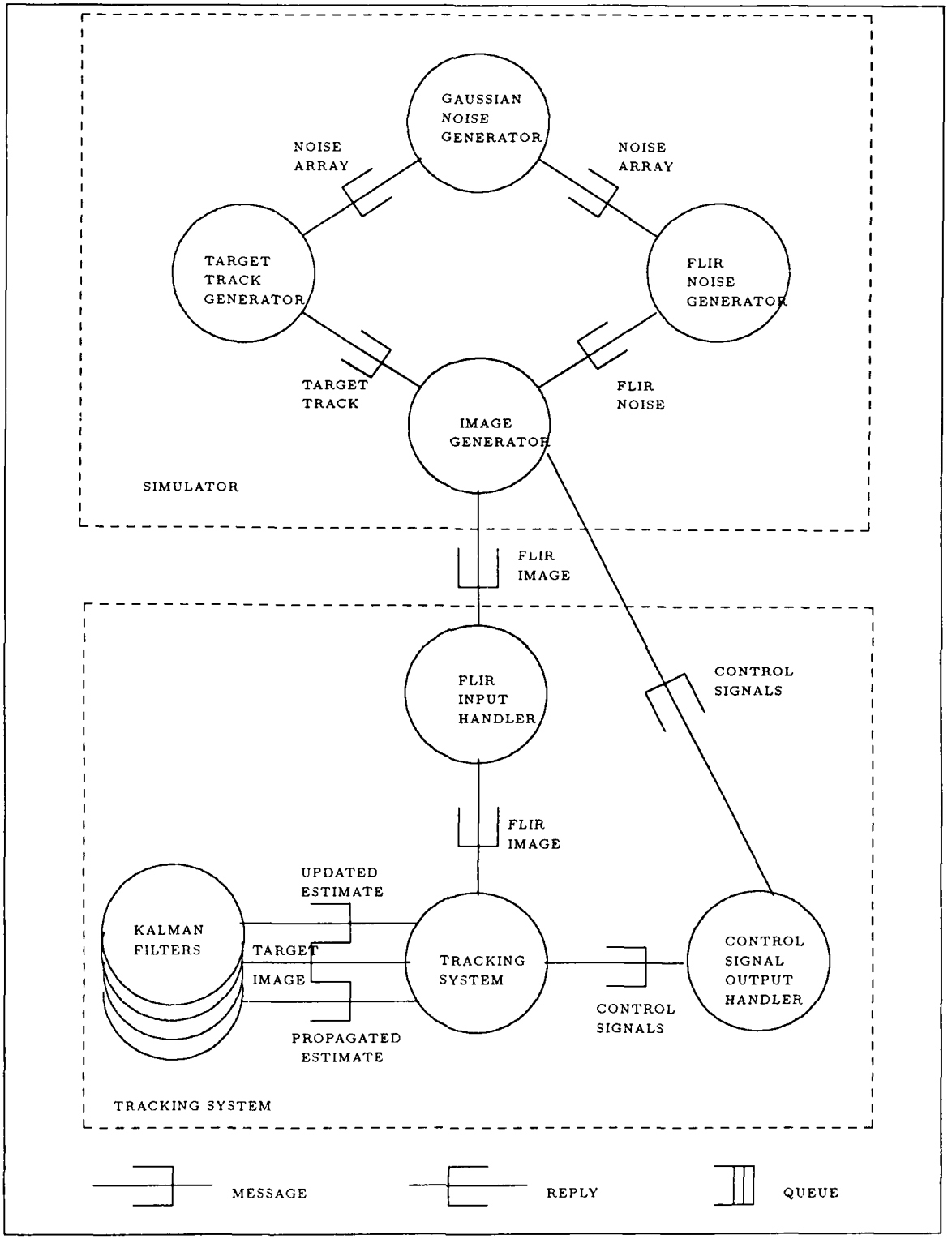

Figure 10. Process Structure Chart

The image generator passes complete simulated FLIR images to the tracking system input handler one at a time. No response is required from the tracking system input handler so the processes are only moderately coupled. However, because the image generator is dependent on the tracking system control output to generate the next image, it must be synchronized with the tracking system and only one FLIR image can be produced at a time. The FLIR image is passed through the input handler to the tracking system.

The communication between the tracking system and the Kalman filters is more complex. The message traffic consists of three independent messages per frame: transfer of the target centroid information from the tracking system to the filters, transfer of the updated state estimates from the filters to the tracking system, and transfer of the propagated state estimates from the filters to the tracking system. None of these messages require replies so the level of coupling is moderate.

The control signals generated by the tracking system are passed to the signal output handler as they are produced. No reply is required and no queueing is possible because only one set of control signals will be in the system at any one time. These control signals are passed to the image generator to complete the next FLIR image and to a data collector for accuracy analysis. In the implementation, this data collector will be the main procedure.

4.4.2 Introduction of Intermediary Processes. Intermediary processes are introduced to decrease coupling between tasks and allow them to run in an asynchronous fashion. This, in turn, leads to better speed-up in a multiprocessor environment. In most cases, these processes are inserted where the tightest coupling occurs. However, additional processes increase run-time overhead, and this concern must be balanced against the increased efficiency that results from decreased coupling. Four intermediary processes were used in this design. The results of these additions and the conversion of all processes into Ada tasks is shown in the Ada task graph, Figure 11.

The area of tightest coupling in the design is between the FLIR noise generator and gaussian noise generator and between the target track generator and the gaussian noise generator. Despite this, no intermediary tasks were introduced here because the relatively short periods of synchronization that result did not justify the overhead of additional

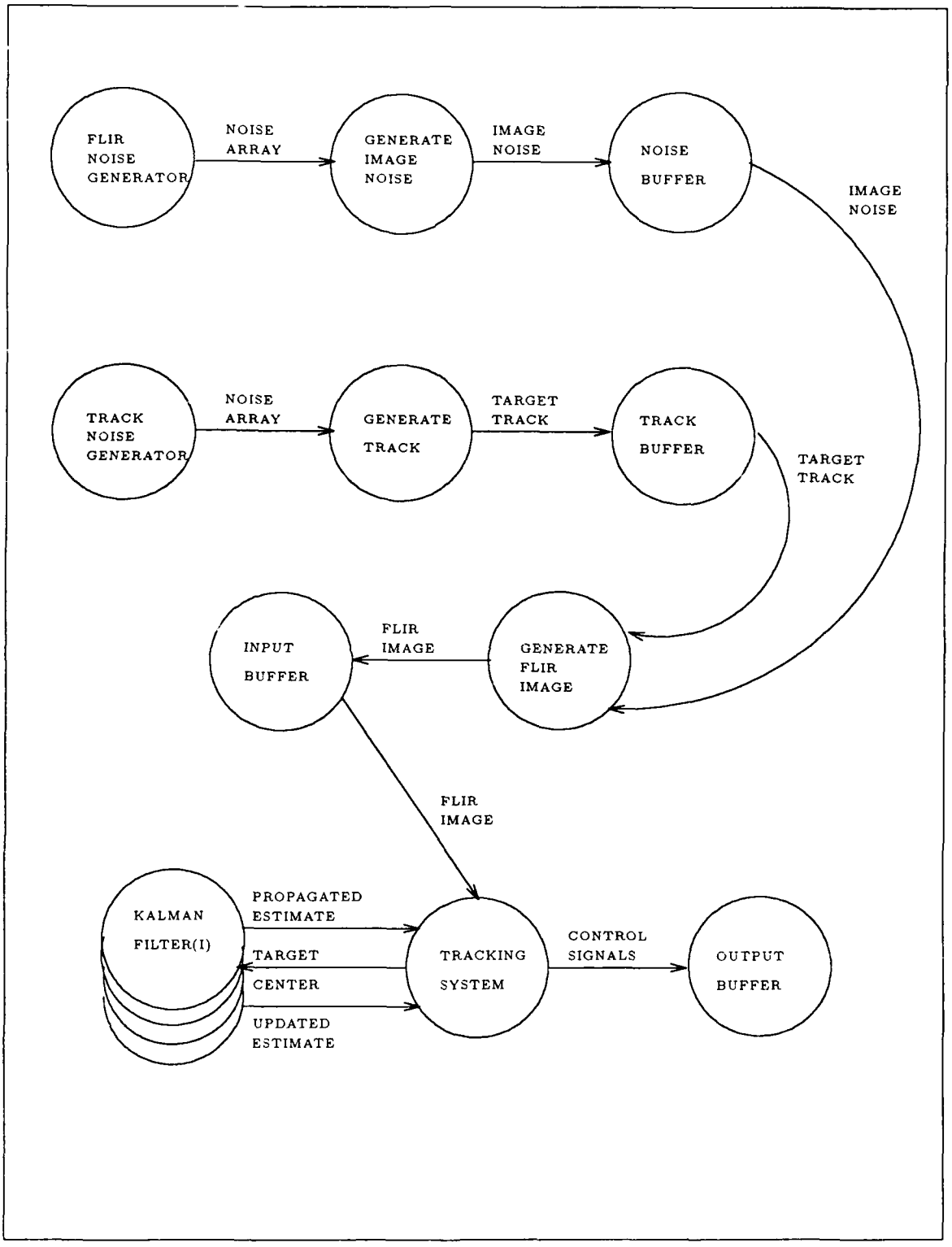

Figure 11. Ada Task Graph

tasks. However, to reduce contention at the noise generator, two noise generator tasks were created, one for the FLIR noise generator and one for the track generator. In addition, no intermediaries were inserted between the tracking system and the Kalman filter banks. In none of the three cases of synchronization between the tracking system and the filters was the calling task able to procede until the called task had completed processing. Therefore, the coupling between the tasks incurred no additional processing delay.

Efficiency was increased by inserting buffer tasks between both the FLIR noise generator and target track generator and the image generator. Because the outputs from the noise and track generators were not in any way dependent on previous outputs, it was advantageous to decouple them from the image generator. In this configuration, the noise and track generators are free to operate at their own speed, and the noise and track buffers serve as queues to store the resulting outputs until the image generator is able to accept them. Buffer tasks were also inserted in place of the input and output handlers. This decouples the tracking system from the operation of the image generator and the data collector.

4.4.3 Introduction of Data Driven Design. Assuming the existence of a concurrent Ada run-time system, the concept of data driven design is relatively easy to incorporate into this design. Within the tracking system, the data flow is continuous, regular, and unidirectional. In each data frame, the FLIR image is input, correlated with the template created in the previous frame, and the resulting target centroid information passed to the Kalman filter bank. This bank produces a propagated state estimate used to create control information and an updated state estimate used in turn by the correlator to create the template for the next frame. This process repeats itself ad infinitum with no required deviations. As a result, this flow of data, in the context of an Ada run-time system, can be used to produce automatic task scheduling.

When no FLIR data is available, the task containing the correlator is in a blocked state. As soon as new data is available, the process unblocks and correlates the new image. It then passes the centroid information to the individual Kalman filters and blocks again. The Kalman filters are waiting in a blocked state for this information. They

unblock, process the propagated and updated state estimates, pass this information back to the correlator, and then block again. This blocking and unblocking action ensures that processes are only consuming computing resources when they have useful work to do. It also ensures that, given sufficient processing resources, each task will be scheduled as soon as the data is available that it needs to proceed. Assuming an efficient run-time system, the result is optimal processor use, superior to that which could be achieved using a static allocation scheme or even a cyclic executive.

4.4.4 Encapsulation of Ada Tasks into Packages. The packaging decisions in this design were made to maximize modularity and the localization of both functions and data structures. This will increase ease of implementation and unit testing as well as result in simpler maintenance of the system. It will also increase the reusability of system components.

Global data declarations, constants, and type definitions are collected into a single master package. The package also contains the four buffer tasks and certain complex number functions which were not available in the system standard math library. The gaussian noise generator, FLIR noise generator, target track generator, and image generator were all encapsulated into separate packages along with their respective local type and variable definitions.

The tracking system task and Kalman filter task type are contained in the same package, while the sequential tracking system routines that are called by these tasks are in a separate package. The division was made to separate the mathematically intensive system subprograms from the operational tasks and increase the maintainability of the two. Finally, several support routines were encapsulated separately to increase their reusability. These include an FFT package, a portable random number generator package, and a generalized matrix operations package.

The authors of the LVM/OOD methodology suggest that these packaging decisions be graphically displayed using an Ada package graph (Nielsen and Shumate, 1988:61-62). This graph is basically a reproduction of the Ada task graph with shading used to show the placement of the tasks in packages. This graph has three problems. First,

it is only possible to show task placement; the placement of subprograms, data objects, and even subordinate tasks cannot not be shown in an Ada package graph. Second, it is impossible to show package dependency. This dependency is a critical issue in both the implementation process and any future maintenance and must be documented. Finally, the Ada package graph gives the false impression that communication occurs between package bodies. Communication may only occur between run-time units (tasks and subprograms), and should not be displayed in a graph meant to show package structure.

To overcome these difficulties, a different method of displaying Ada packages is provided here. It will be refered to as as a package structure chart. Packages are described using symbols from (Booch, 1983). This symbology makes it possible to show the placement of data objects and subprograms, as well as tasks. The package is shown as a rectangle with "windows" in its left wall for each of the operations or objects which it exports. Operations are represented by rectangles in that wall, and objects are shown as ellipses. Tasks are represented as parallelograms with rectangles in the left wall to document entries. Independent subprograms, those not encapsulated in packages, are shown as rectangles with enclosures at the top representing their specifications. Arrows drawn between packages are used to show package dependencies. The package structure chart for the Kalman filter tracking system is shown in Figure 12.

4.4.5 Decomposition of Large Tasks. At this point in the design, all of the process abstraction is complete. The purpose of this step is to take those of the independent tasks which are too complex to implement efficiently in the task body and decompose them further. The method of layered virtual machines is used for this purpose. Each function is repeatedly divided into subcomponents until each part is simple enough to be implemented directly.

In the Kalman filter tracking system, the buffer tasks and the gaussian noise generator required no further division. The FLIR noise, target track, and image generators as well as the Kalman filter and correlator tasks are all computationally complex and required further decomposition. The results are documented in the following figures. Nielsen and Shumate suggest using structure charts to display the interactions between higher and

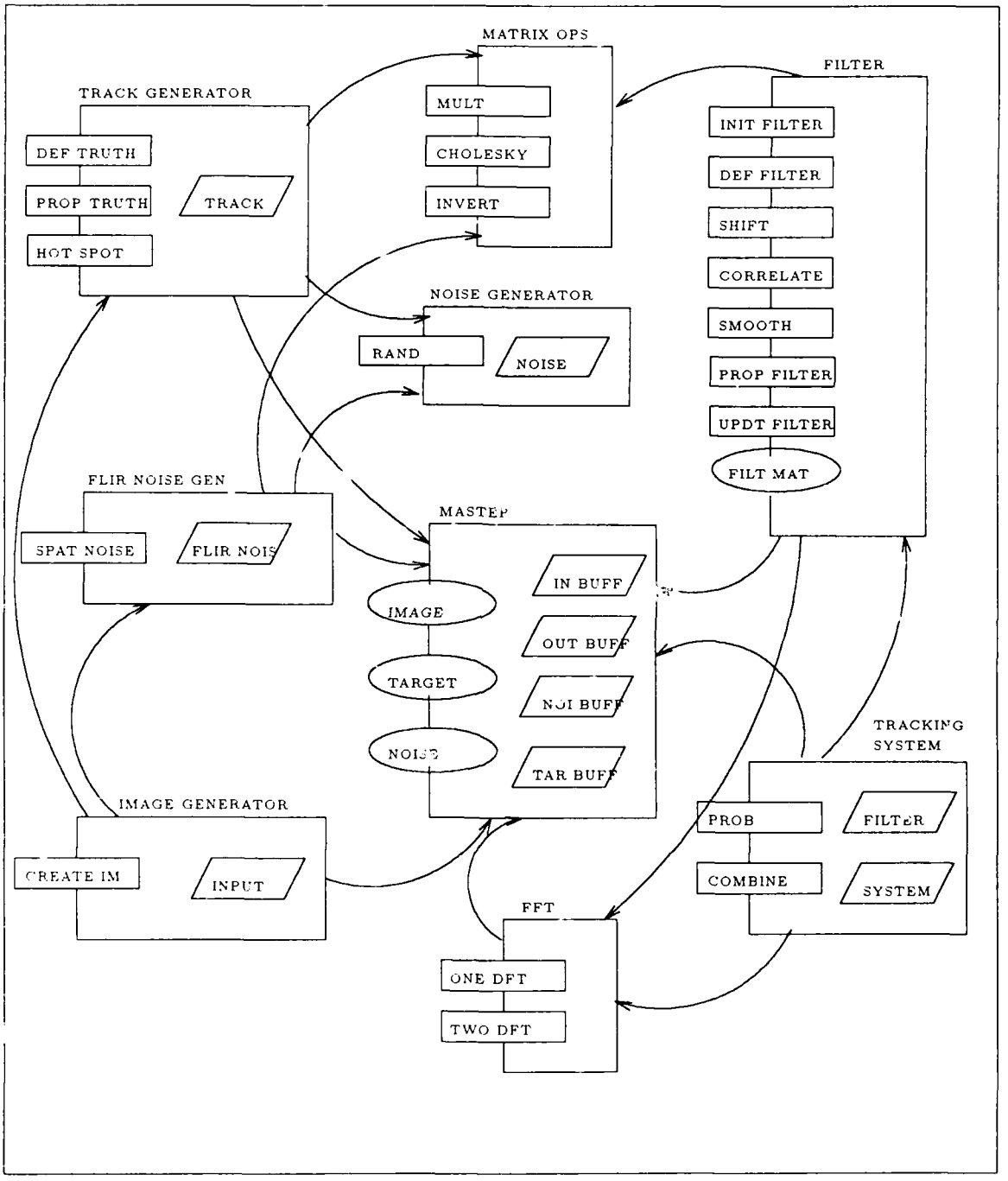

Figure 12. Package Structure Chart

lower level virtual machines (Nielsen and Shumate, 1988:133-136). While these charts are adequate for showing sequential operations, they provide no means for showing interaction between lower level subprograms and independent tasks.

In order to show this communication, the standard structure chart has been modified in two ways. First, an additional symbol has been added in addition to the widely recognized rectangle. While the rectangle will continue to represent calls to subprograms, a circle will now be used to show a call to an independent task. In this case, the data objects displayed on the connecting lines will be the parameters passed during a rendezvous rather than subprogram specifications.

The second difference from standard structure chart is that the levels of the chart no longer represent the physical subordination of subroutines in the corresponding implementation. While the layers still show an abstraction of a problem from its most difficult to its simplest functions, the resulting subprograms and tasks need not be subordinate and may actually be compiled separately from the "parent routine." This change is in keeping with the separate compilation abilities of the Ada language, but it requires severe limits on the use of global variables. Because global variables detract from the maintainability and understandability of the program and should be avoided whenever possible, this restriction does not present a problem.

The purpose of the track generator task, shown in Figure 13, is to generate a simulated target track mathematically. The first step is to define a truth model which will be used to propagate the position of the target from one time frame to the next. After the truth model is defined, the initial position of the target must be calculated. The steps involved in this process are shown in Figure 14. Once the simulation begins, the task must continue to produce target images for the number of frames in the simulation. This involves computing target parameters, projecting them into a three hot spot image, updating the current position using the truth model, and passing the resulting target information to the track buffer. This process is shown in Figure 15.

The purpose of the image noise generator is to create random noise to be inserted into the simulated FLIR image. This noise is used to simulated real-world background

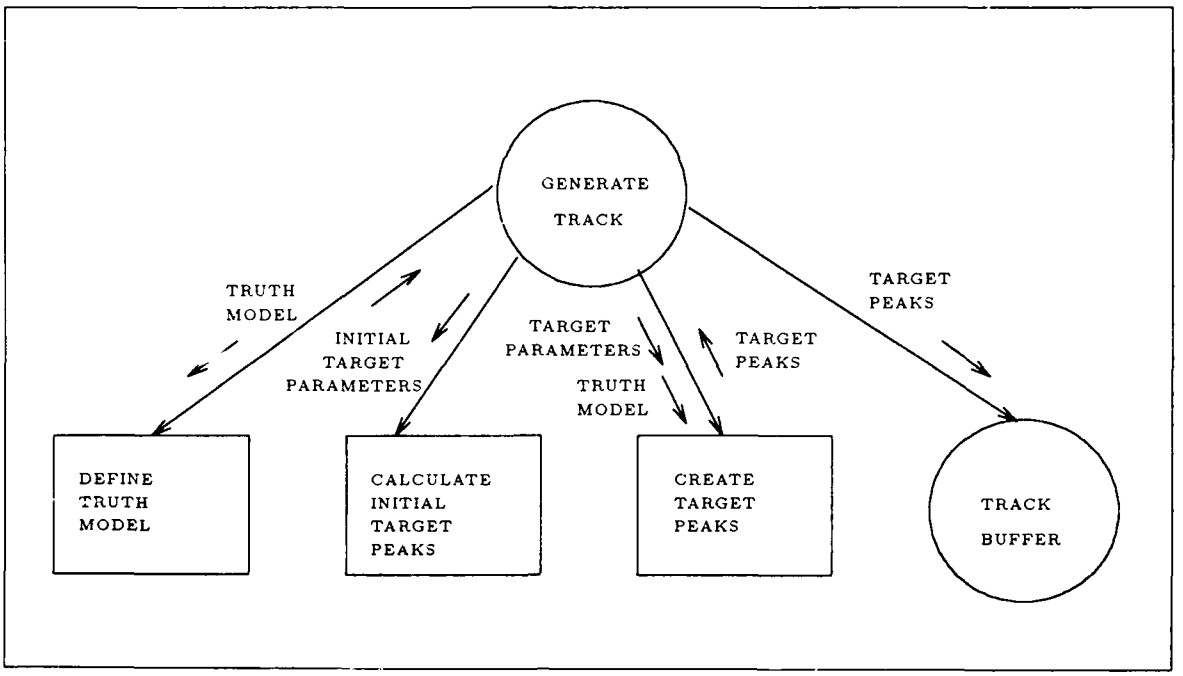

Figure 13. Track Generator

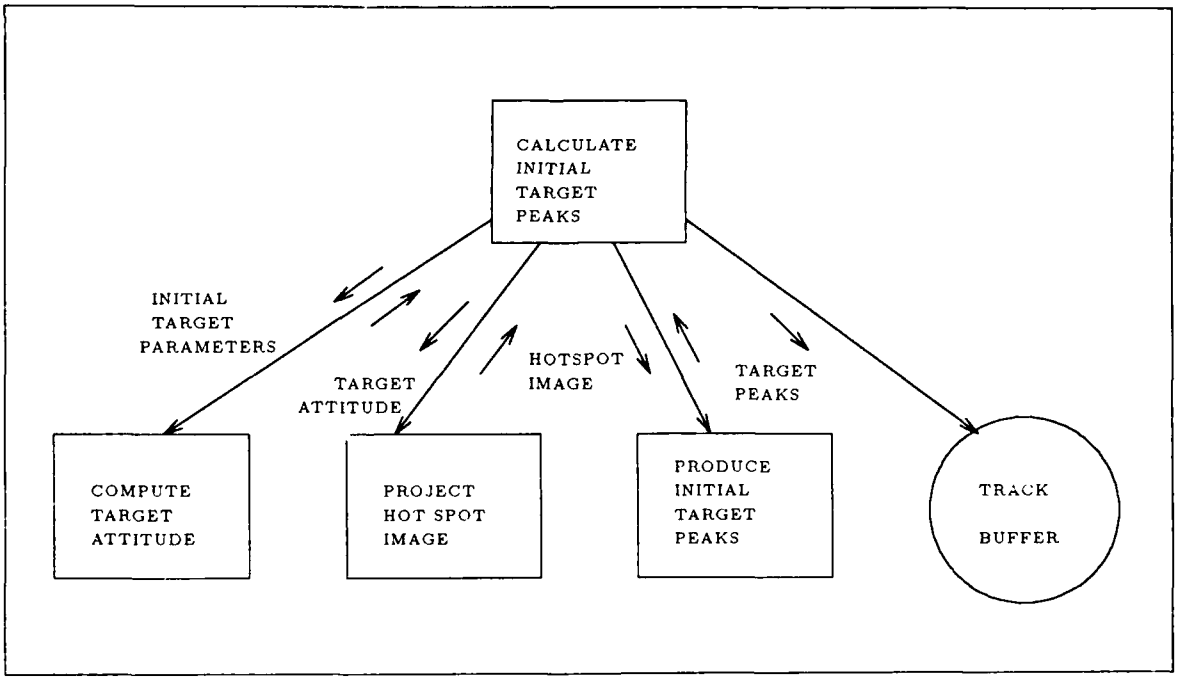

Figure 14. Calculate Initial Target Position

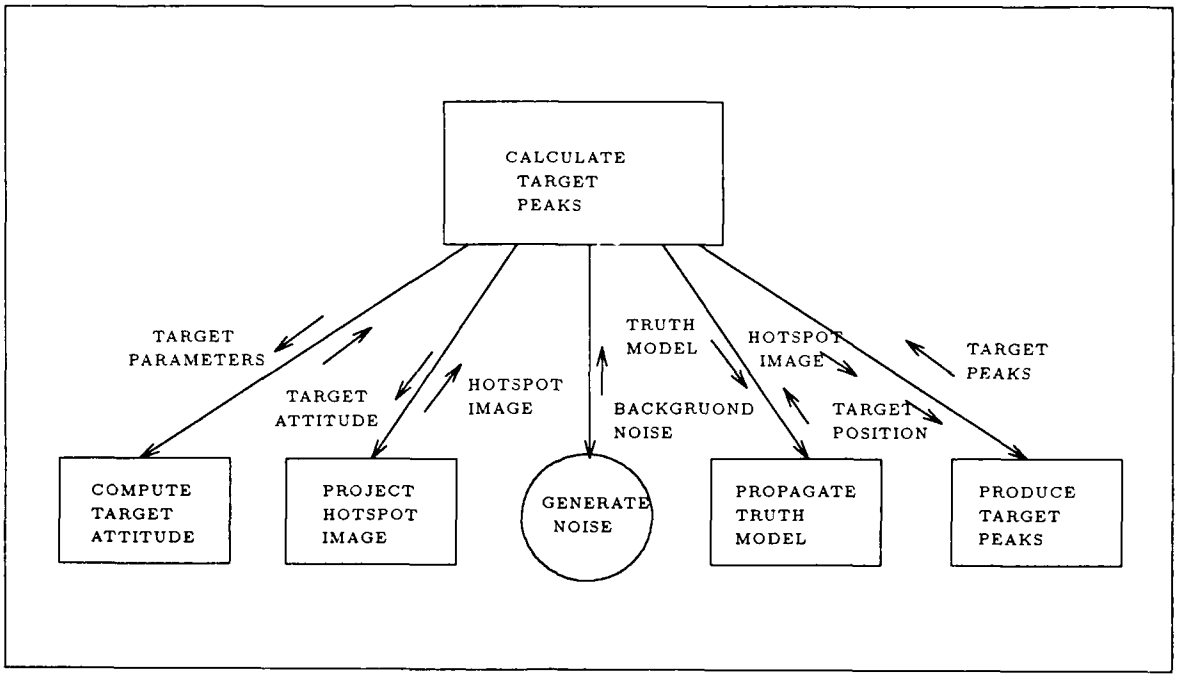

Figure 15. Calculate Current Target Position

noise and internal FLIR noise (such as thermal noise, dark current, etc). The generator creates two types of noise: random, to represent uncorrelated interference, and correlated, to represent noise patterns with some, but not perfect, correlation. To create the correlated noise, a spatial noise correlation coefficient matrix is set up using second nearest neighbor correlation, and a Cholesky decomposition is performed on the matrix. After these initial steps, the noise generation process begins. One array of each type of noise is created for every frame. This process consists of obtaining two arrays of random numbers from the Gaussian noise generator, adjusting one of them to create correlated noise, and passing the two arrays to the noise buffer. The complete image noise generator is shown in Figure 16.

The purpose of the input generator is to combine the outputs from the track and image noise generators, adjust them for the current position of the pointing mechanism, and combine them into a single array that simulates the image input from the FLIR sensors. This process must be accomplished for each frame. The complete input generator is shown in Figure 17. At this point in the design, it became obvious that the Create FLIR Image process was a candidate for additional parallelization. The operation is homogeneous and

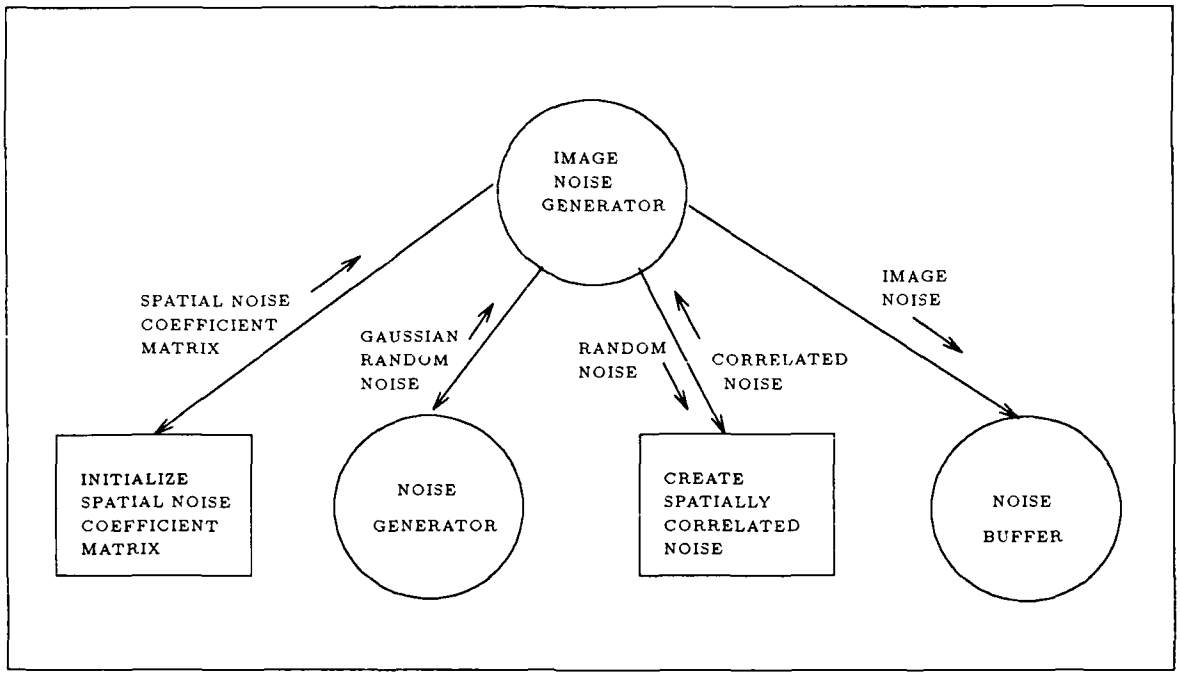

Figure 16. Image Noise Generator

is performed on a large data set (the 24-by-24 image array). It is also computationally intensive. It was therefore parallelized using the concept of partitioning.

Although there are four separate Kalman filter processes, each of them is identical in structure. That common structure is shown in Figure 18. The differences in operation are accounted for by the filter parameters which are input at the start of the simulation. Each filter begins processing by using these parameters to initialize its state transition matrix and discrete noise matrix. In addition, the filter state vectors must be initialized and the first state transition propagated to the tracking system. The initialization process is shown in Figure 19. After the first frame, each filter waits to receive target centroid information from the tracking system. It then uses this information to update the current state estimate and propagate the future state estimate. These results are passed to the tracking system at the appropriate points in the frame. This process continues until the end of the simulation.

The tracking system acts as the master process in the real-time portion of the program. The high level structure is shown in Figure 20. The first step is the initialization

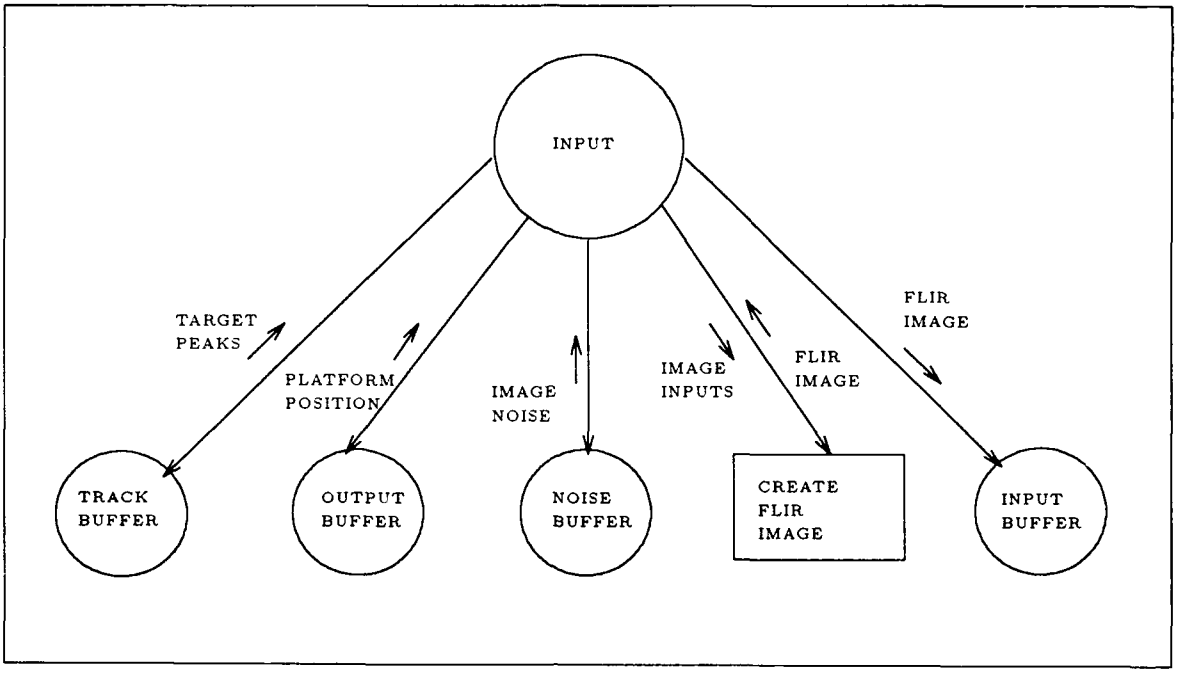

Figure 17. FLIR Image Input Generator

process shown in Figure 21. This process acquires the target using its image in the first FLIR input. Once the target is acquired, the tracking system repeats the same functions for each data frame. After the FLIR image is accepted from the input buffer, it is correlated with the target template and the resulting centroid information passed to the Kalman filters. When the updated state estimates and residuals are received from the individual Kalman filters, the new filter probabilities are calculated and used to combine the updates into a single multiple model estimate. This information is used to create the new template as shown in Figure 22. Once the propagated state estimates are received, they too are combined using the same probabilities to create a single multiple model estimate. This propagated estimate is used to calculate the control inputs which are then passed to the output buffer.

The main process is the controller for the entire simulation. Its sole purpose is to accept the simulation parameters (either from an input file or from the user at the terminal), use these parameters to initialize each of the other tasks in the system, and then act as the platform pointer by accepting the controller outputs from the tracking system. These

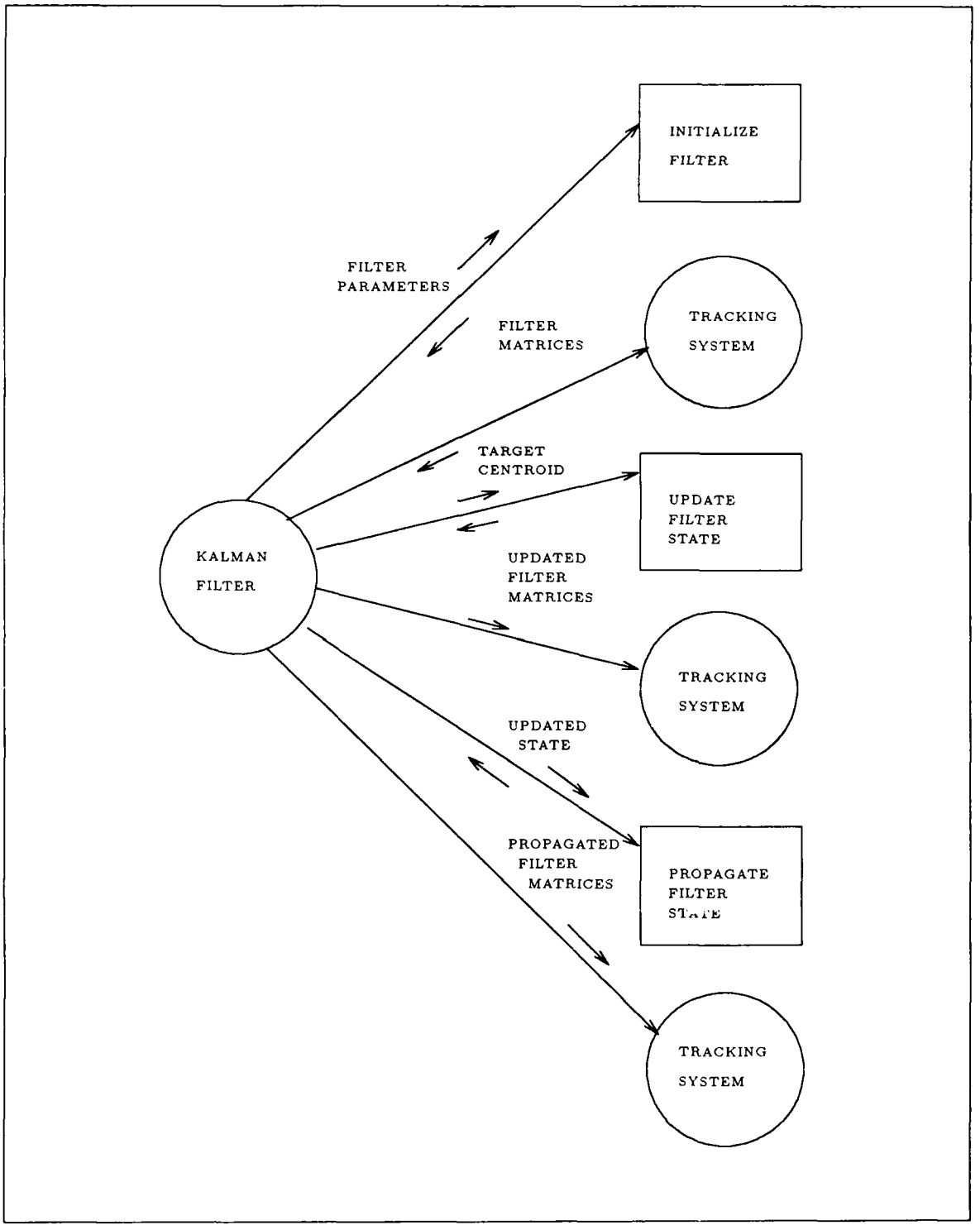

Figure 18. Kalman Filter

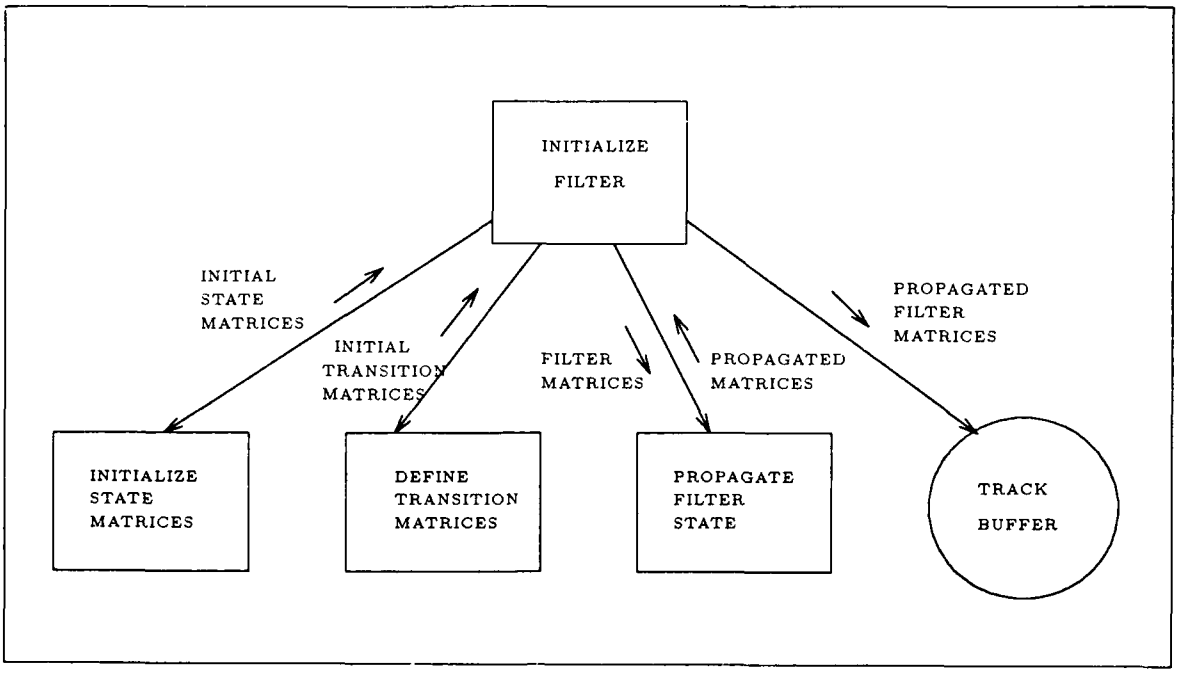

Figure 19. Kalman Filter Initialization

outputs are trapped and used later to validate the performance of the system.

At this point, each of the tasks identified in Section 2 that required decomposition has been described using the Layered Virtual Machine concept. For purposes of brevity, the lowest level structure charts have not been included here. The detailed design is now complete.

4.5 Chapter Summary

This chapter outlined the development of the Kalman filter tracking system from initial problem analysis to the final detailed design, using the concepts developed in Chapter 3. The analysis was based on research conducted at the Air Force Institute of Technology and on a sequential FORTRAN program developed to support that research. However, to keep the research manageable, only a representative subset of that program was included in the design.

The system selected for implementation was based on four Kalman filters fed by a single enhanced correlator. Raw FLIR inputs enter the system in the form of a single 8-by-8

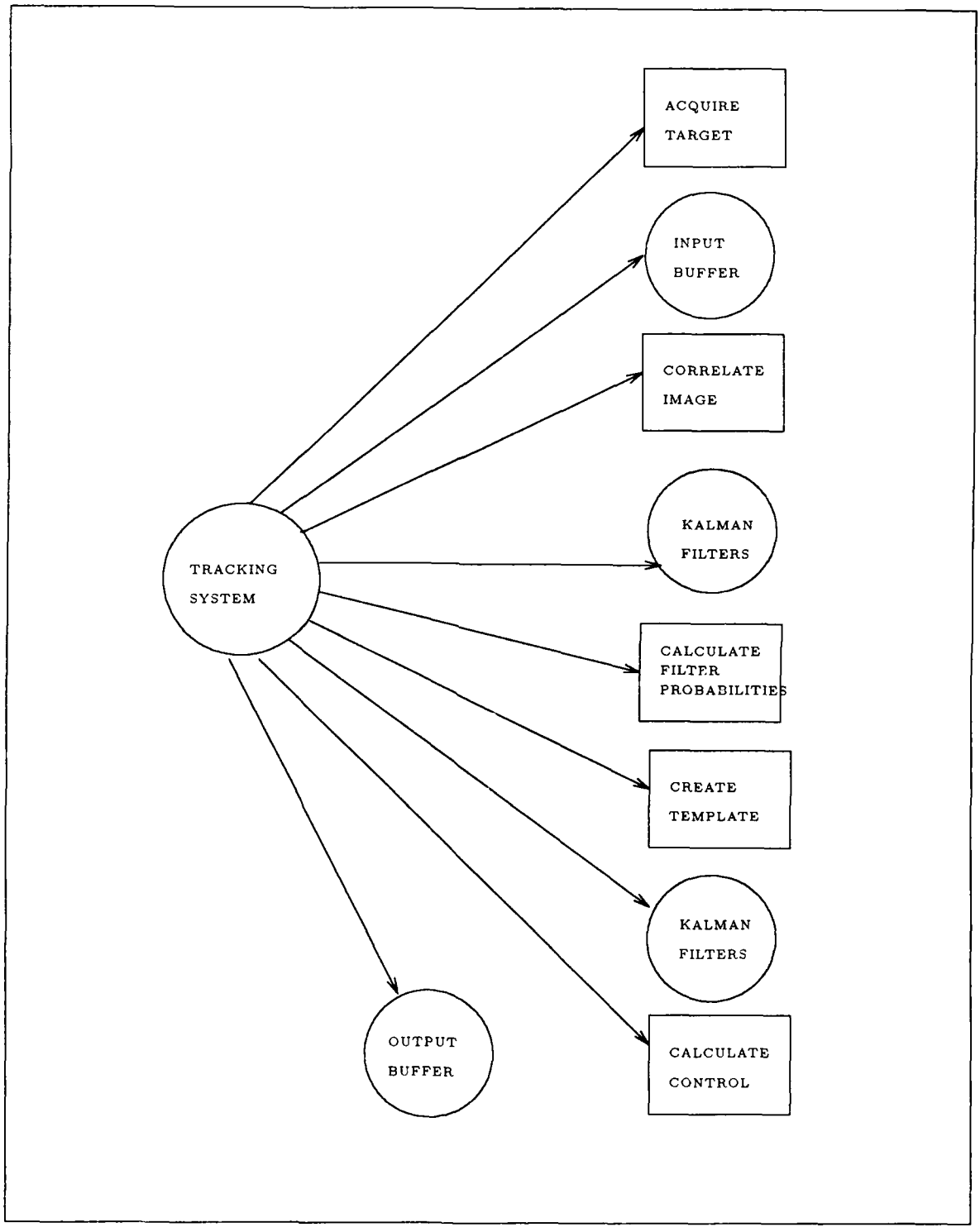

Figure 20. Tracking System

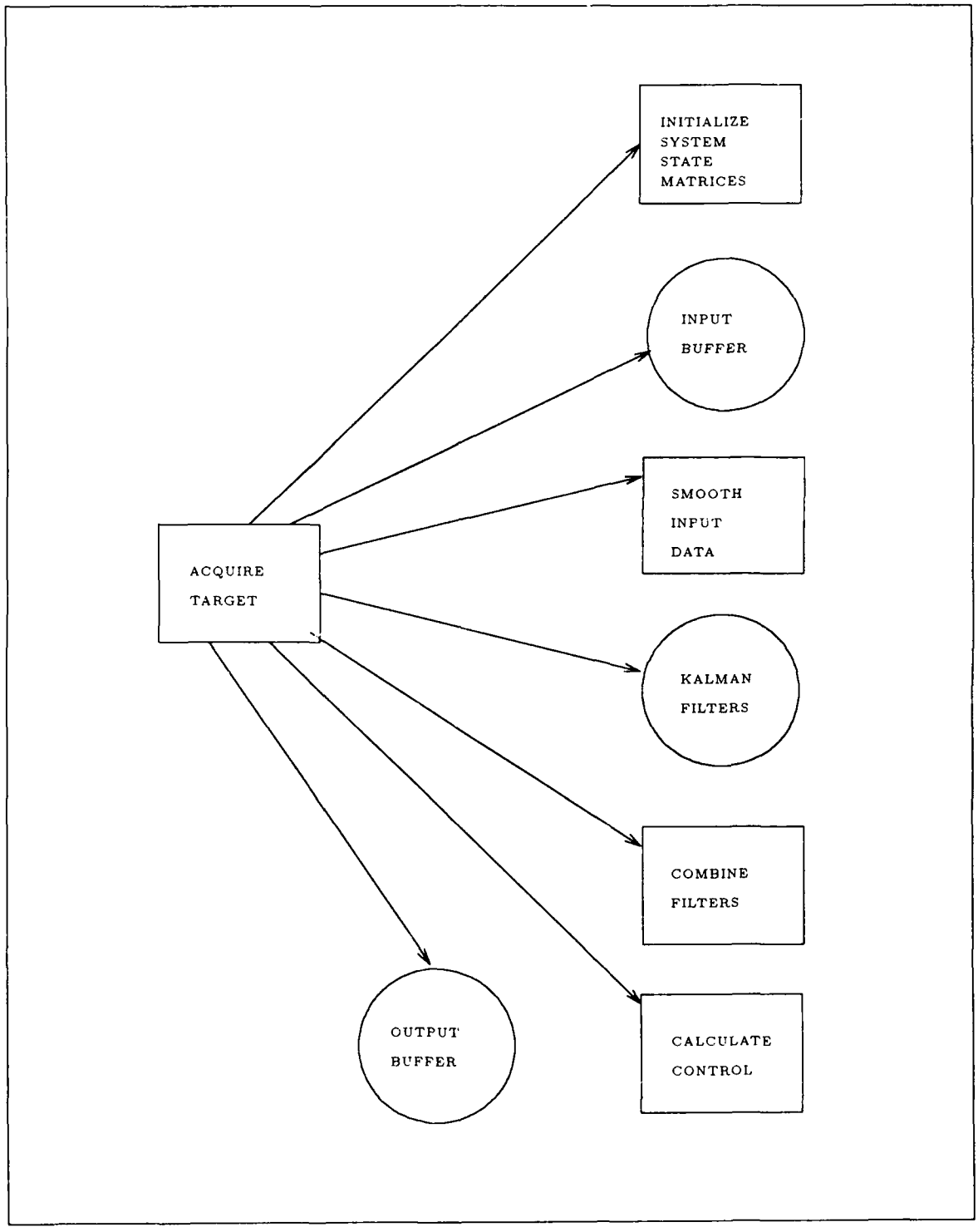

Figure 21. Tracking System Initialization

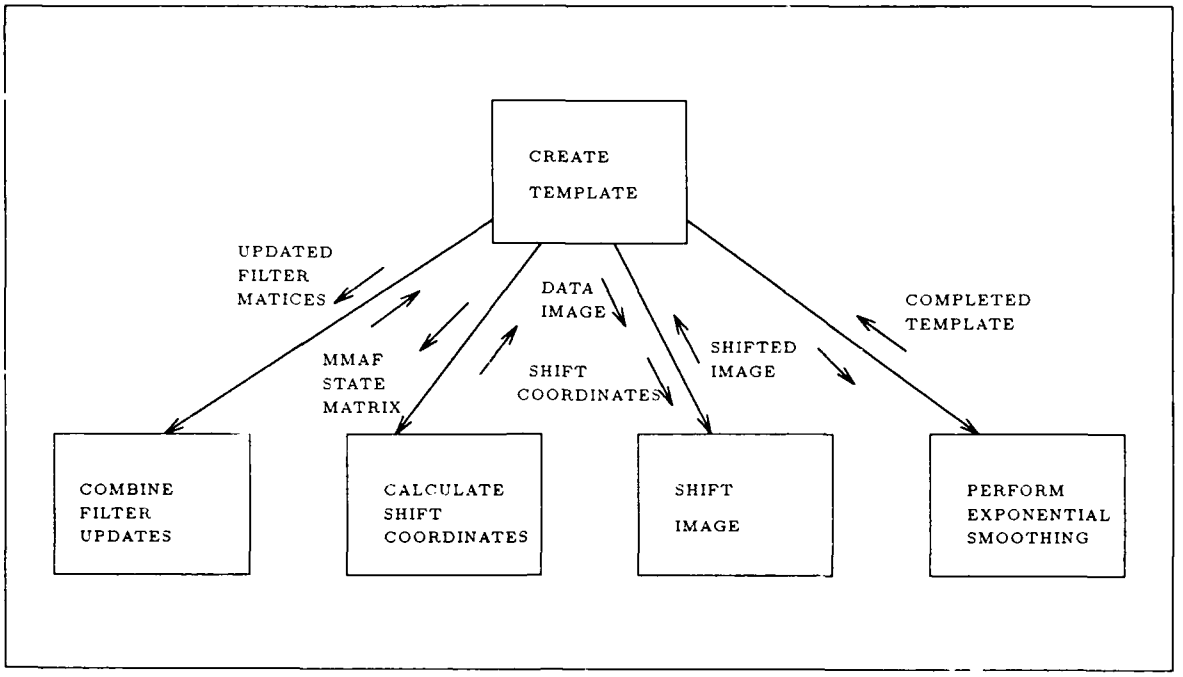

Figure 22. Template Generation

array of pixel intensities. The enhanced correlator uses this information to produce a single offset "pseudo-measurement" which is used by the linear Kalman filter bank to update the current system state estimate and propagate a new one for the next frame. In addition to the tracking system proper, a simulation program is necessary for test purposes. It consists of a random noise generator, a target track generator, and a routine which combines the outputs of the other two into a simulated FLIR image.

Decomposition of the problem was attempted at three levels. At the system level, the possibility of pipelining was explored. However, data dependencies made the enhanced correlator an essentially sequential operation and no parallelization was possible for the tracking algorithm at this level. Relaxation was used at this level to parallelize the simulator. At the Kalman filter bank level, relaxation was used to model each filter as a separate process. Finally, at the support routine level, partitioning was used to parallelize certain computationally intensive matrix operations.

After completion of the problem decomposition, the system and detailed design were accomplished using the LVM/OOD methodology. In two places, the method was altered

to accommodate the problem better. First, a package structure chart was used in place of the Ada package graph. Second, the standard structure chart used to document the decomposition of large tasks was replaced with one that showed task interaction in addition to subprogram dependency. With the detailed design complete, the next step was to translate it into Ada code. That process is described in Chapter 5.

V. Implementation and Testing

The purpose of this chapter is to discuss the implementation and subsequent testing of the Kalman filter tracking system design described in Chapter 4. It begins with a description of the hardware and software tools used in the implementation. Section 2 reviews the development of certain reusable support routines, including: a random number generator, a Gaussian noise generator, a parallel matrix multiply routine, and a parallel Cholesky decomposition routine. Section 3 outlines the two part development process used for this project and describes how testing was integrated into each step of the process. Finally, Section 4 discusses the importance of timing analysis in a concurrent software environment and explains how this analysis was used to increase the efficiency of the Kalman filter tracking system.

5.1 Development Environment

This section describes the hardware and software tools used to implement the Kalman filter tracking system. The system was implemented on an Encore Multimax 320, a tightly-coupled multiprocessor architecture based on the National Semiconductor 32332 processor. This processor runs at 15 MHz and is rated at 2.0 MIPS. Each processor is also equipped with its own 64K cache memory. Main memory is shared and is eight-way interleaved to reduce memory contention. Memory is reached via an extremely high-speed bus. The bus operates off a system clock running at 12.5 MHz, resulting in a data bandwidth of 100 Mbytes/Sec. The Multimax 320 at the Air Force Institute of Technology is configured with 16 processors and 32 Mbytes of main memory (Encore, 1987).

The operating system is UMAX, which is a derivative of UNIX. The system operates under a single copy of UMAX which is accessable by all of the processors simultaneously. UMAX implements the concept of multi-threading, allowing it to support multiple, simultaneous streams of control. Interprocess communication occurs through shared memory rather than explicit message passing. The operating system also supports process migration, making dynamic load balancing possible (Encore, 1987).

The Encore Ada run-time environment makes true concurrent Ada possible using the task construct. Rather than using some single-processor interleaving scheme, Encore Concurrent Ada allows the user to specify the number of independent processes desired. Tasks are then scheduled on a first-come/first-served basis from a single queue when a process becomes available. Tasks priorities are supported through the use of multiple task queues, one for each priority level. The user has no explicit control over task allocation or scheduling. Once he sets the number of processes, these processes are assigned to processors by the operating system. Tasks are assigned to available processes by the Ada run-time system (Encore:1988).

5.2 Reusable Support Routines

The burgeoning demand for software necessitates exploring every possible method of increasing productivity. One such method is the reuse of proven components. Other engineering disciplines have always made extensive use of this concept. However, "in the past, software reuse has been hindered by the fact that it was quite difficult to isolate program parts that could be reused in the context of another project" (Roubine, 1986:1). Even if certain routines could be isolated, there was no guarantee they would be portable because of the use of non-standard programming language features. Thus, valuable programming effort was often used "reinventing the wheel."

Today's large software developments provide extensive opportunities for component reuse. This is true even in the case of specialized embedded systems (Roubine, 1986:1). In order for software to be reusable, it must have four necessary characteristics. First, each routine must present simple, clear interfaces. Second, it must be portable to a variety of different hardware suites. Third, each component must be made as reliable as possible through full unit testing. Finally, the components should be adaptable to different environments (Roubine, 1986:2).

One of the greatest assets of the Ada language is its support of reusability. The package construct provides a convenient method of encapsulating software components, and the package specification contains simple, clear interfaces unclouded by implementation details. The Ada language standard ensures the portability of components to any machine

with a validated Ada compiler. Reliability can be maintained through independent testing of Ada routines and the use of exception handlers. Finally, the Ada generic construct provides the capability for producing adaptive software.

With reusability in mind, the first part of the software development effort in this thesis was devoted to producing several general software components including: a random number generator, a matrix inversion routine; and parallel matrix multiply and Cholesky square root routines. The two parallel support routines are based on the concept of partitioning as discussed in Chapter 3. A description of the development of each component follows. In addition to these routines, a fast Fourier transform routine was adapted for this problem from one developed by (Kobel and Martin, 1986).

5.2.1 Random Number Generator. In order to simulate the noise portion of the FLIR sensor input, it is necessary to have a random number generator. Because as many as 100,000 random numbers are necessary to simulate a single five minute target track, the random number generator needs to have an extremely long period and be free from sequential correlation. In addition, portability demands that a routine supplied by the system not be used.

In order to meet the given constraints, a portable random number generator was implemented in Ada using an algorithm by Donald Knuth from (Press and others, 1982:196-197). The period of this routine is practically infinite, and it has no sensible sequential correlation. The choice of constants and the length of the shuffling array is arbitrary (Press and others, 1982:196).

5.2.2 Matrix Inversion Routine. The inversion of matrices is required in the update portion of the Kalman filter and by the routine that calculates the relative probability of each filter in the filter bank. A method for parallelizing matrix inversions is presented in (Fox and others, 1988), but the tracking system uses the routine infrequently and the largest matrix inverted is 2-by-2 elements. This did not justify the overhead involved in parallelization, and a sequential routine was implemented instead. This routine uses Gauss-Jordan elimination, and the algorithm comes from (Press and others, 1982:28-29).

5.2.3 Parallel Matrix Multiply Routine. The Kalman filter tracking algorithm uses matrix multiplication extensively. In addition, the simulation program multiplies matrices as large as 64-by-64 elements. The extensive computation necessary to multiply such large matrices, coupled with the ease with which the algorithm can be parallelized, more than outweighs the added overhead of multi-tasking.

The form of the parallel algorithm is suggested by (Quinn, 1988). The greatest speed-up can be gained by parallelizing the outermost of the three multiplication loops because the number of operations in this loop is $O(N^3)$. If there are N total columns to calculate in the result matrix, T tasks are used with each task calculating N/T columns. Ignoring memory contention, the speed-up will be nearly linear (Quinn, 1988:132).

The two input matrices and the result matrix are global to the matrix multiply procedure. All tasks have equal access to these data structures with memory contention being resolved by the run-time system. This structure is ideal for a shared memory machine.

Instead of using a fixed number of tasks, the main procedure spawns tasks dynamically, based on the size of the input matrices. The number of tasks spawned is determined using a granularity constant which is set at compile time. The total number of elements in the result matrix is calculated, and the number of tasks spawned is equal to the number of elements divided by the granularity constant. If there is an uneven number of columns, any extra columns are processed by the main procedure. This method acts as a form of load balancing by matching the number of tasks to the overall workload. A suitable granularity constant can be determined using empirical timing data. This data is obtained by making several test runs on the target hardware using differing constants and selecting the one that best balances multi-tasking overhead and parallelization.

5.2.4 Parallel Cholesky Decomposition Routine. In the simulation program, both the definition of the target truth model and the calculation of spatial noise for the simulated FLIR sensor inputs requires taking the square root of a matrix. In the case of the spatial noise generator, this matrix is 64-by-64 elements. One efficient means of taking the square root of a matrix is the Cholesky decomposition algorithm. This algorithm is computationally intensive ($O(N^3)$) and can benefit from parallelization.

The Cholesky square root can be generated sequentially using the following equations from (Maybeck, 1979:371) where $i = 1, 2, \ldots, n$ and A_{ij} represents the element in row i column j:

$$\sqrt{A_{ij}} = \begin{cases} (1/\sqrt{A_{jj}})[A_{ij} - \sum_{k=1}^{j-1} \sqrt{A_{ik}}\sqrt{A_{jk}} & j = 1, 2, \ldots, i-1 \\ (A_{ii} - \sum_{k=1}^{i-1} \sqrt{A_{ik}^2})^{1/2} & j = i \\ 0 & j > i \end{cases}$$

Parallel implementation of this algorithm is complicated by its recursive nature. In order to calculate element j in row i, the corresponding element in all previous rows must have already been computed. This precludes the simultaneous calculation of all rows as in the matrix multiplication routine. George et al call these recursive dependencies "precedence relations." These precedence relations fit into one of three categories: Type 1: task a must finish before task b can begin, Type 2: task a must finish before task b can finish, and Type 3: task a must begin before task b can begin. Precedence relations place an important constraint on the run-time system scheduler. Tasks cannot be scheduled randomly; they must be scheduled in the order their precedence relations demand (George et al, 1986:167).

The routine implemented in this thesis uses an algorithm analogous to what (George et al, 1986) describe as "Row-Cholesky." The input matrix is global to the Cholesky procedure and all operations are done in place. Each task processes the elements in one row in order and replaces the input element with the result. The recycling tasks are drawn from a ready pool and passed the number of the row they are to compute. After processing their designated row, they return to the task pool to be used again. Tasks are terminated once all rows have been calculated.

These tasks operate under a type 3 precedence relationship. The task operating on the ith row cannot begin before the task operating on the $i-1$st row. Tasks operating on all rows previous to i must always be a minimum of one element ahead of the task operating on the row that follows. This synchronization is provided by a controller task which will not start a task to process row i until it receives word from the task operating on row $i-1$ that it has computed its first element.

The number of tasks in the task pool is not static. The routine dynamically spawns tasks to match the workload in a manner similar to the matrix multiply routine. The number of tasks spawned equals the number of rows in the input matrix divided by a granularity constant. This constant is set by the programmer at compile time in the same manner as the matrix multiply routine.

This routine depends on the task scheduler using a single FIFO queue to schedule tasks for all processors. This is the only way to ensure that precedence relations are met and that a task operating on row i is not scheduled until all tasks operating on previous rows are either operating or completed. This dependency on the order of scheduling may seem to violate the spirit of non-deterministic tasks, but (Cohen, 1988) observes that such dependency does not violate any language rules. It does not cause erroneous execution, nor does it lead to incorrect order dependence. It does, however, limit the portability of the routine to run-time systems with the same or a functionally equivalent scheduling order. This is the price that must be paid for parallelizing algorithms with precedence relations.

5.3 System Implementation and Testing

This section describes the process used to implement the Kalman filter tracking system using the design documented in Chapter 4. Both top-down and bottom-up strategies were used. A top-down approach was used to build a skeleton of the system to outline and test system interfaces. These interfaces were used as a basis for a bottom-up construction of the final system. Testing was completely integrated with the implementation, occurring at the end of each phase of the development.

5.3.1 Top-down Development. The first phase of the top-down approach began with the coding of the packages shown in Figure 12. Global type descriptions were declared in the package specification of the package in which they were first referenced. In addition, simulation model constants were declared in package Master. With these high level data descriptions in place, a skeleton of the complete system was developed using the processes shown in Figure 11 as a basis.

Each of the processes was initially implemented as a separate task with the exception

of the Kalman filter and the Gaussian noise generator. The Kalman filter was coded as a task type and an array of four Kalman filters was declared to complete the filter bank. The Gaussian noise generator was also coded as a task type and separate noise tasks were declared in the track and FLIR noise generator packages. Each task declaration was complete with all of its entries and their corresponding parameters. The task bodies were coded as shells containing only the *entry* calls and *accept* blocks from the final implementation. An output statement was included before and after each *entry* call and *accept* block describing the current state of the task (eg., "TASK INPUT IS CALLING TASK INPUT BUFFER ENTRY PUT IMAGE" or "TASK OUTPUT BUFFER IS WAITING TO ACCEPT ENTRY GET CONTROL FROM TASK TRACKING SYSTEM"). All variables local to the task bodies were declared and those used for communication parameters were initialized to zero.

Once the coding was complete, the skeleton was compiled using the already established order of compilation (based on the task dependencies identified in Figure 12). The compilation served several purposes. First, it provided a first check of package dependence. The compiler would have flagged any attempted use of a type declaration or communication with a task resident in a package other than those in the current compilation unit's *with* clause. Second, the actual parameters in the *entry* calls were checked against the formal parameters in the *accept* statements and any type mismatch would have been caught. Finally, the compiler would have flagged any improper parameter mode with a warning (eg., if a variable used as an *in out* parameter had not been initialized).

Once compiled, the skeleton system was executed on a single processor. The execution served two purposes. First, it caused the elaboration of all of the task and object declarations and activation of all the tasks. One problem was detected here. The run-time system puts a limit of 10K bytes on the size of each task stack. Because large matrix objects in several of the tasks exceeded this limit, these tasks aborted during elaboration resulting in a TASKING ERROR exception. This problem was circumvented by recoding these tasks as task types and using a representation specification to force the system to allocate more stack space. A single task object was then declared of each task type.

The second purpose was to check the system for possible deadlock conditions. Al-

though no actual data was processed or passed between tasks, all rendezvous took place and were documented by the output of the task state descriptions described earlier. The resulting record of system operation provided clues as to the state of the system at the point at which deadlock occurred. This information was useful in reordering *entry* calls to eliminate the deadlock problem.

Despite the usefulness of the system state output technique, it does have two limitations. First, there is no guarantee of what an output device driver will do when a concurrent system deadlocks. Some messages may be lost and the resulting state record may not be complete. Second, even a deadlock-free system at this point does not guarantee continued correct operation once the skeleton has been filled in with all the required processing. Because of the non-deterministic nature of the tasking model, additional processing load may lead to a different task scheduling order. This differing order may present opportunities for deadlock that do not exist in the skeleton. Only careful analysis of the system design and the results of tests can lead to a dead-lock free system. Nonetheless, until better tools for following tasking flow of control are made available, the use of a system state description log is still one of the best methods of finding concurrent software communication errors.

The second stage of the top-down part of the development involved adding the subprogram specifications to their respective package specifications and subprogram stubs to the corresponding package bodies. The subprogram stubs contained all of the local variables but no operations. Subprogram calls were added in the appropriate places in the task bodies using the detailed design as a guide. The packages were then recompiled.

This second compilation served the same purpose for the subprograms that the first did for the high-level tasks. Package dependencies were checked again, and the formal and actual parameters in the subprogram specifications and calls were compared for type mismatches. After compilation, the skeleton system was executed again to force elaboration of the subprogram local variables.

5.3.2 Bottom-up Development. The completed system skeleton provided a framework for further development by documenting the interfaces between all of the high-level processes. Knowledge of these interface points formed a basis for further development of

the system from the bottom up. The reusable routines described in Chapter 2 were completed first. They were each tested independently using specifically designed test cases.

The lowest level subprograms were coded next using the reusable routines and other atomic operations. These subprograms were also tested independently, where possible, but this was made difficult by a lack of test cases. Test cases were constructed for some routines by capturing inputs and outputs at various points in the FORTRAN program, but the results were not always comparable due to the differing ways the languages and machines performed certain mathematical functions. The fact that random numbers play such a large part in the simulation also made replicating results difficult.

Once the lower level subprograms were completed, they were integrated at an intermediate level into four separate subsystems: the simulated noise and track generators, the simulated image generator, and the tracking system proper. Separate test harnesses were written to test the noise and track generators first. The outputs from these two subsystems were compared to outputs from the corresponding parts of the FORTRAN program. After these two subsystems were successfully tested, they were integrated with the image generator and the three were tested as a unit. The resulting data array was again compared to the corresponding output from the FORTRAN program. By this time, round-off and computation differences had become significant, and it was hard to determine whether differences in the results were due to these factors or some error in the Ada implementation. Finally, the completed simulation subsystem was integrated with the tracking system portion. The control outputs were compared to those of the FORTRAN program.

Several errors were discovered during the testing process. Most of them were numeric in nature and could be traced to the use of the wrong variable or an incorrect operation in one of the complex mathematical equations. Two errors were also the result of using uninitialized variables on the right side of an expression. Apparently, the run-time system simply used whatever values were in those locations at the time without providing some warning of their misuse. In addition to these errors, the frequent use of unconstrained arrays as formal parameters resulted in several constraint violations going unnoticed until run time. Having to debug these errors at run time was the price that was paid for the use of generalized routines.

The difficulty in finding many of these errors was increased by the fact that the Encore run-time system would not propagate unhandled exceptions out of tasks. As a result, when a NUMERIC ERROR or CONSTRAINT ERROR occurred within a task without an exception handler, the run-time system would simply hang without an error message. Often this required moving the code to another development environment (most often to a DEC Ada system on a VAX 11/780) in order to determine what the error was. The portability of Ada was a real advantage here, as the entire Kalman filter tracking system could be ported with minimal changes to the code (mostly involved with relinking the math library). The DEC Ada system was often very helpful in locating the source of the error. The Verdix debugger was most often useless in these situations. After testing was completed and the system was fully integrated, the code was returned to the Encore Multimax for timing analysis.

5.4 Timing Analysis

One additional step in any parallel software implementation should be timing analysis. Once all of the code is running "correctly," the system should be analyzed from a timing perspective to determine if any modifications could result in more efficient task scheduling and communication and, consequently, faster run-times. In this case, such analysis was made relatively simple by two factors. First the Multimax uses a single, global clock. In effect, each processor "sees" the same time which makes it very convenient to compare process start and stop times across processors. Second, the Encore Ada run-time system supports the serialization of I/O in a multiprocessor environment. This makes it possible for all the processes to write to a single log file without overwriting each other's messages.

For purposes of timing analysis, calls to the system clock were inserted at various points in the Kalman filter tracking system software. The first call was included in the body of the Master package. It was used to set a start time to be used as a reference point by the rest of the system. In addition, clock calls were included at the beginning and end of each iteration of the major tasks as well as at certain key rendezvous. The start time reference was subtracted from the results of each of these calls to produce a module

time-stamp relative to the system start. This time stamp and its corresponding system state reference was then output to a single system log for later review.

Review of the system log was very helpful in finding two places where the tracking system could be made more efficient. First, timings showed that the propagate and update operations contained in the Kalman filter bank did not require sufficient CPU time to justify two separate rendezvous with the tracking system task (see Figure 18). Therefore, both the update and propagate operations were performed in sequence and then a single entry call was used to transfer both the updated and propagated state and covariance matrices to the tracking system.

The second enhancement came in the tracking system task itself. In the original design (see Figure 20), the template for the next frame was calculated first, using the updated state information from the filter bank, and then the propagated information was used to calculate the control outputs. However, because the FLIR image generator is dependent on this control information to generate the next frame's FLIR image, this ordering resulted in needless delay to the image task. The order of the operations was reversed so that control outputs were calculated first and sent to the output buffer, and then the next frame's template was generated. This resulted in more efficient parallel operation of the tracking system and the FLIR image generator tasks.

The use of task priorities was also examined to determine if elevating the priority of some of the tasks would increase efficiency. Frequently called tasks, the Gaussian noise generator and the buffer tasks, were set at a higher priority to ensure them greater access to system resources. However, different priority levels had a negligible effect on system operation, and all of the tasks were subsequently returned to the same priority.

5.5 Chapter Summary

The Kalman filter tracking system was implemented on an Encore Multimax 320 configured with 16 processors and 32 MBytes of main memory. The software environment consisted of a Verdix Ada compiler targeted to the Multimax and Encore's concurrent Ada run-time system.

As a first step in the implementation process, several reusable routines were developed to support matrix and random number operations, including: a random number generator and Gaussian noise generator, a matrix inversion routine, a parallel matrix multiply routine, and a parallel Cholesky decomposition routine. These programs were individually tested and encapsulated in packages to enhance portability.

After completion of the support routines, the Kalman filter tracking system was implemented using both top-down and bottom-up strategies. A top-down approach was used first to build a skeleton of the system consisting of only the top level tasks and their corresponding *entry* calls and *accept* blocks. This skeleton was compiled and run to check package dependencies, task communication, and for possible deadlock conditions. In the second stage of top-down development, subprogram specifications and body stubs were added to the skeleton, which was then compiled and run again.

With the top-down portion of the implementation complete, the skeleton served as a basis for constructing the rest of the system from the bottom up. The lowest level subprograms were coded first, followed by an intermediate integration into four separate subsystems: the FLIR noise generator, the target track generator, the simulated FLIR image generator, and the tracking algorithm. As a final step, these four subsystems were combined into the completed tracking system. Testing occurred throughout the implementation process at the end of each phase. Whenever possible, data from the corresponding part of the FORTRAN program was used to test each completed module.

After system integration and testing, timing analysis was used to determine if any modifications could be made to the system to increase the efficiency of task scheduling and communication. This analysis resulted in two changes being made to the system: elimination of one of the Kalman filter entry calls, and a reordering of the correlator operations. With timing analysis complete, and the resulting modifications made and tested, this phase of the project was complete. The source code from this research is archived at the Air Force Institute of Technology in the Department of Electrical and Computer Engineering. For information, contact Dr Thomas Hartrum. The next step was *operational testing*. The results of that testing are documented in Chapter 6.

VI. Results and Conclusions

This chapter presents an analysis of the results of this research in the context of the specific objectives outlined in Chapter 1. Specific conclusions on each objective are included in the respective sections. Section 1 is an analysis of the operational testing of both the Kalman filter tracking algorithm and its associated simulator. Section 2 reviews the effect of using the parallel software engineering guidelines described in Chapter 3 to design and implement a non-trivial problem. A specific analysis of the current weaknesses of the method is also included. Section 3 is an analysis of the strengths and weaknesses of using Ada for a parallel software development. The final section presents recommendations for future research in all three of the preceding areas.

6.1 Analysis of the Tracking System Implementation

One objective of this research was to use parallelism to speed up the execution time of both the Kalman filter tracking algorithm and its associated simulation program significantly. In order to gauge the results in this area, several types of timing analyses were done on the operational system. The Multimax architecture and the Ada language made this analysis relatively simple. As stated earlier, the Multimax has a single system clock. Therefore, allowances for processor clock variance (as in the case of loosely-coupled architectures) are not necessary. In addition, the Ada language provides direct access to the system clock via the resources of package Calendar.

Execution times were recorded by making a call to function Clock at the start of the main procedure and again at the end, subtracting the start time from the end time, and writing the result to a log. This type of analysis ignores the initial elaboration and activation time of the dependent tasks; however, sufficient iterations of the main program were executed to make this time insignificant in calculating overall speed-up results. Speed up was calculated using the definition found in (Stone, 1987:120) of time for serial execution divided by time for parallel execution.

Five types of timing tests were performed on the operational system. The simulation program was tested first at only the highest level of parallelization (using sequential matrix

operation routines) and then again with the parallel support routines included. Similarly, the Kalman filter tracking algorithm was tested at the highest level of parallelization and then again with the parallel support routines linked in. Finally, the simulator and tracking algorithm were integrated and the entire system was tested.

In the case of the simulation program, significant speed-up was gained from parallelization. Without the benefit of the parallel support routines, a speed-up of approximately 1.5 was achieved using two processors. Additional processors provided no benefit at this level of testing. This was as expected because the simulation program consists of only three processes at this level (the noise, track, and image generators) and none of them were completely independent. With the use of parallel support routines (including matrix multiplication, Cholesky decomposition, and parallel image creation), a speed-up of 1.6 was achieved with two processors, 2.3 with three processors, and 2.7 with four processors. Considering the unparallelizable parts of each process, the data dependencies between processes, and the inherent parallel overhead, this was an entirely acceptable result.

The results in the case of the Kalman filter tracking algorithm were not as positive. Neither the high-level parallelization nor the addition of the parallel support routines resulted in any significant speed-up. Further timing analysis showed that this result was attributable to two major factors. The first was the inability to parallelize the enhanced correlator portion of the tracking algorithm. The CPU time required to process this part of the system was more than 30 times that of each of the individual linear Kalman filters. The independent computation of each of the filters was of little benefit in relation to the much larger sequential load carried by the single correlator process. The second factor limiting speed-up is that the matrix operations contained in the tracking algorithm are not computationally intense enough to overcome the added overhead of the parallel support routines. These results make it clear that, unless a method can be found to decrease the processing time of the correlator process significantly, this particular version of the Kalman filter tracking algorithm is not suitable for parallel implementation.

The final testing of the complete system also brought disappointing results. These were expected at this stage because the computational loading of the enhanced correlator outweighed that of the rest of the system combined. In addition, the requirement that the

FLIR image generator wait for the previous frame's control outputs before generating the next frame's data introduced additional delay into the system.

In summary, the actual implementation of the Kalman filter tracking algorithm and its associated simulator brought mixed results. The parallelization of the simulation program brought good results. However, the tracking algorithm implementation identified a serious sequential bottleneck in the form of the enhanced correlator. The effects of this bottleneck were severe enough to limit the effects of parallelization on the tracking system as a whole.

6.2 Analysis of the Parallel Software Guidelines

Another objective of this research was to define a comprehensive set of parallel software development guidelines and test it in the implementation of a non-trivial problem. The results in this area were favorable. The guidelines worked well, providing for a clear decomposition of the Kalman filter tracking system into independent processes and an efficient mapping of those processes to the Ada task construct. Nielsen and Shumate's design methodology proved to be an effective means of documenting a parallel design with the inclusion of the changes noted in Chapter 4. However, the implementation brought to light two limitations of the parallel software guidelines as presented in Chapter 3.

The first was the discovery in the implementation phase that the enhanced correlator was a significant roadblock to parallel speed-up with the tracking algorithm. Although some problem was expected based on the results of the initial decomposition (it was clear then that no method was available to parallelize the correlator process on a macro level), the magnitude of the bottleneck was unexpected. Clearly, it would have been better to recognize the severity of the problem earlier in the design process or even in the initial analysis. While the factors limiting decomposition efficiency were discussed early in the guidelines, no real method was provided for discovering the relative magnitude of these factors.

In a parallel environment, such a method cannot be based on standard algorithmic complexity analysis. It must include knowledge of actual module execution speeds and

actual time delays for task allocation, scheduling, and communication. Whether such information could be determined during analysis and design, without some degree of code test cases, is uncertain. Also, the availability of such information presupposes knowledge of the specific implementation hardware. This knowledge may not be available in the early stages of the design. The form of such a complexity analysis method, as well as where in the development cycle it should fit, are areas which require further study.

A second limitation to the guidelines was the lack of a method for specifying independent process interaction graphically. This would have been very helpful in identifying possible deadlock situations. It would have also provided a means of showing data dependency among tasks which would have helped in analyzing task coupling. A very complex problem, involving many layers of tasks, would be impossible to comprehend without some graphical display of task interaction.

6.3 Analysis of Ada as a Parallel Systems Development Language

The final objective of this effort was to determine the adequacy of the Ada language for parallel software development through the implementation of a real-world problem. The implementation of the Kalman filter tracking system highlighted a very important distinction between the adequacy of the language itself (as described by MIL-STD 1815A) and the adequacy of the tools (compilers, debuggers, run-time systems, etc.) currently available to support it. Therefore, these two issues will be discussed separately.

The Ada language proved to be an excellent means of abstracting a parallel problem. The task construct is ideal for encapsulating independent processes. In addition, the rendezvous provides the means for both task synchronization and communication without resorting to some form of machine dependent parallel language constructs. The ability to spawn tasks dynamically was also useful as a load balancing tool. It made it possible for a generalized routine (such as the matrix multiply routine) to vary the number of tasks spawned based on the size of the particular data structure being operated on.

Aside from the task construct, other features of the language made implementation more productive as well. The use of the package construct greatly assisted in modular

development. The use of generics and unconstrained arrays made it possible to construct general support routines capable of operating on a wide range of data forms. Finally, the standardization of the language was itself an advantage. Because of the difficulties encountered with the debugging tools, four different Ada environments resident on four different hardware architectures were used during the development process. The use of Ada made it possible to move modules of code freely between these environments on a frequent basis with a bare minimum of changes.

While the Ada language provided ideal support for the implementation of the system, the tools available to support it still have a great deal of maturing to do. Although the compiler used on the Encore was validated, it soon became obvious that development in a parallel environment is as dependent on the correct operation of the run-time system as it is on the compiler generating valid executable code. Several problems were encountered with the Encore run-time system, and they made program testing very difficult.

Because the Ada language standard does not cover run-time system implementation, there is no validation capability available for run-time system operation. Therefore, as was the case with previous languages, the user is dependent on vender testing to ensure a valid system. A method of central validation was one of the major reasons Ada was developed, and serious consideration should be given to adding run-time system standards to the current language standard.

The development tools available for this project were also inadequate. The compilers used were very slow, especially toward the end of the project when long lists of dependent packages had to be recompiled because of minor code changes. The symbolic debugger was next to useless in a multitasking environment and did not support concurrent multitasking at all. There were also no tools available to monitor the execution of the parallel processes, making task analysis very difficult. Finally, the documentation available on the concurrent aspects of the run-time system was all but non-existent. This lack of development support, coupled with Ada's proven portability, reinforces the idea that Ada code development should be done on a machine specifically designed for the purpose (Booch, 1986; Caruso, 1985), and the resulting code ported to the final target machine upon completion.

6.4 Suggestions for Future Research

This section contains suggestions for future areas of study based on the results of this research. The first four relate to continued efforts into parallelizing the Kalman filter tracking system. Items five and six are suggested as additions to the parallel software engineering guidelines begun here. The final item pertains to the Ada language.

1. One possible method of decreasing the enhanced correlator bottleneck would be to parallelize the fast Fourier transform operation. The FFT and inverse FFT in the correlate portion of the tracking system task currently account for 80% of the total CPU time required to process that task. If significant speed-up could be achieved in this area, it could make this version of the Kalman filter tracking algorithm viable for parallel implementation. Studies on the parallelization of FFTs have been done by (Briggs et al, 1987) and (Aliosio et al, 1987).

2. If the speed of the enhanced correlator cannot be significantly increased, another option is to use a different form of the Kalman filter tracking algorithm. The extended Kalman filter (EKF) approach has also been investigated at the Air Force Institute of Technology (Maybeck and Rogers, 1983; Maybeck and Suizu, 1985), but it was abandoned in favor of the single enhanced correlator approach due to its high level of computational loading in a single system. However, the EKF approach off loads much of the correlator's workload to the individual, filters which are parallelizable. Therefore, this approach may result in better efficiency in a parallel software environment.

3. Any embedded implementation of the Kalman filter tracking system would have to deal with the issue of fault tolerance. Although the Ada exception handling mechanism is available to handle many types of faults, the use of a distributed system adds the possibility of hardware, communication, and distributed processing faults for which the current language definition does not provide. Initial research into the area of distributed fault tolerance in Ada has been begun by (Reynolds et al, 1983), but much more work is necessary.

4. A real-time embedded avionics system operates differently from a shared memory mainframe such as the Multimax in such areas as process allocation, scheduling, and

communications, memory management, and other critical run-time issues. The Kalman filter tracking system should be ported to an embedded distributed processor group as soon as one becomes available that supports the full Ada tasking model.

5. Research needs to be done into some method of software complexity analysis for parallel systems. This method must go beyond standard algorithm complexity analysis to include such issues as communication and tasking overhead and particular hardware issues such as execution speed. Study must also be done to determine where this method would fit into the software development cycle described in Chapter 3 and when, and to what degree, knowledge of the specific implementation hardware should be included in the process.

6. Some method for specifying complex task relationships needs to be developed, This method should incorporate issues such as task dependency, data dependency, and task communication. The method must be able to detect possible deadlock situations. It must also be able to specify multiple layers of interacting tasks.

7. Software tools are needed to support parallel development in Ada. These should include a symbolic debugger capable of operating in a concurrent environment through several levels of tasks and subprograms. Also included should be a tool which allows the developer to observe the allocation, scheduling, and communication of tasks during program execution. Graphic representation of system operations would be very helpful in this area.

Bibliography

1. Aloisio, G. and G. C. Fox, et al "A Concurrent Implementation of the Prime Factor Algorithm on the Hypercube," unpublished Caltech report C3P - 468 (1987).
2. Baker, T. P. "Parallel Processing and Ada Tasks," *Application to Ada Higher Order Language to Guidance and Control.* NATO Advisory Group fro Aerospace Research and Development, 1986.
3. Bolz, Richard E. "Introduction to Ada," *Application to Ada Higher Order Language to Guidance and Control.* NATO Advisory Group fro Aerospace Research and Development, 1986.
4. Booch, Grady. *Software Engineering with Ada.* Menlo Park: Benjamin/Cummings Publishing Company, Inc, 1983.
5. Booch, Grady. "Ada Development Gets a Lift from an Intelligent Tool Set," *Electronic Design*: 111-114 (August 7, 1986).
6. Briggs, William L. and Leslie B. Hart, et al. "Multiprocessor FFT Methods," *SIAM Journal of Scientific and Statistical Computing, Vol 8 No. 1*: s27-s42 (January 1987).
7. Burger, Thomas M. and Kjell W. Nielsen. "An Assessment of the Overhead Associated with Tasking Facilities and Task Paradigms in Ada," *Ada Letters, Vol 7 No 1*: 49-58 (January/February 1987).
8. Caruso, Denise "Development System Breaks Productivity Barrier," *Electronics*: 36-40 (July 8, 1985).
9. Cherry, G. W. *Parallel Programming in ANSI Standard Ada.* Reston: Reston Publishing, 1984.
10. Chow A. and M. Feridun. *A Method for Partitioning Real-Time Ada Software for Distributed Targets* September 1986. Contract N00014-85-C-0796.
11. Cline, Carolyn L. and Howard Jay Siegel. "Augmenting Ada for SIMD Parallel Processing," *IEEE Transactions on Software Engineering, SE-11 No 9*: 970-977 (September 1985).
12. Cohen, Norman H. "Dependence on Ada Task Scheduling is Not Erroneous," *Ada Letters, Vol 8 No 2*: 77-83 (March/April 1988).
13. Cornhill, Dennis. (a) "Four Approaches to Partitioning Ada Programs for Execution on Distributed Targets," *Proceedings of the 1984 IEEE Conference on Ada Applications and Environments.* 153-162. Silver Springs: IEEE Computer Society Press, 1984.
14. Cornhill, Dennis. (b) "Partitioning Ada Programs for Execution on Distributed Systems," *IEEE Proceedings of the International Conference on Data Engineering.* 364-370. New York: IEEE Press, 1984.
15. Eastport Study Group. *Summer Study 1985: A Report to the Director of the Strategic Defense Initiative Organization*, December 1985.

16. Encore Computer Corporation. *Multimax Technical Summary* 726-01759 Rev D. March, 1987.

17. Encore Computer Corporation. *VADS Release Notes Concurrent Ada (B 1.0)* January 18, 1988.

18. Flynn, Susan and Edith Schonberg, et al. "The Efficient Termination of Ada Tasks in a Multiprocessor Environment," *Ada Letters, Vol 7 No 7*: 55-76 (November/December 1987).

19. Fox, G. and M. Johnson, et al. *Solving Problems on Concurrent Processors, Volume 1* Englewood Cliffs: Prentice Hall, 1988.

20. George, Alan and Micheal T. Heath, et al. "Parallel Cholesky Factorization on a Shared-Memory Multiprocessor," *Linear Algebra and its Applications, 77*: 165-187 (1986).

21. Hutcheon, A. D. and A. J. Wellings. "Ada for Distributed Systems," *Computer Standards and Interfaces, 6*: 71-81 (1987).

22. Knight, John C. and John I. A. Urquhart. "Fault Tolerant Distributed Systems Using Ada," *Proceedings of the AIAA 4th Conference on Computers in Aerospace*. 325-330. New York: AIAA Press, 1983.

23. Kamrad, J. Micheal "Real Life Considerations of Ada Runtime Organizations for Real-Time Applications," *Proceedings of the IEEE/AIAA 6th Digital Avionics Systems Conference*. 472-476. New York: AIAA Press, 1984.

24. Lane, Debra S. and George Huling, et al. "Implementation of a Real-Time Distributed Computer System in Ada," *AIAA Proceedings of the Fourth Conference on Computers in Aerospace*. 325-330. New York: AIAA Press, 1983.

25. Lerner, Eric J. "Parallel Processing Gets Down to Business," *High Technology*: 20-28 (July 1985).

26. Livingston, Dennis. "Parallel Machines take on Supercomputers," *High Technology*: 26 (July 1985).

27. Maybeck, Peter S. *Stochastic Models, Estimation, and Control, Volume 1* Orlando: Academic Press, Inc, 1979.

28. Maybeck, Peter S. "The Kalman Filter – An Introduction for Potential Users" Class handout distributed in EENG765, Stochastic Estimation and Control I. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, January 1988.

29. Maybeck, Peter S. and Daniel E. Mercier. "A Target Tracker Using Spatially Distributed Infrared Measurements" *IEEE Transactions on Automatic Control, AC-25 No. 2*: 222-225 (April 1980).

30. Maybeck, Peter S. and Steven K. Rodgers. "Adaptive Tracking of Multiple Hot-Spot Target IR Images," *IEEE Transactions on Automatic Control, AC-28 No. 10*: 937-943 (October 1983).

31. Maybeck, Peter S. and Robert I. Suizu. "Adaptive Tracker Field-of-View Variation Via Multiple Model Filtering" *IEEE Transactions on Aerospace and Electronic Systems, AES-21 No. 4*: 529-539 (July 1985).

32. McGee, B. *Software Engineering Library - DCDS Summary Description Volume 1* January 1987. Contract DASG60-85-C-0047.

33. Mundie, David A. and David A. Fisher. "Parallel Processing in Ada," *Computer, 19*: 20-25 (August 1986).

34. Nielsen, Kjell W. and Ken Shumate. "Designing Large Real-Time Systems with Ada," *Communications of the ACM 30*: 695-715 (August 1987).

35. Nielsen, Kjell W. and Ken Shumate. *Designing Large Real-Time Systems with Ada.* New York: Multiscience Press, Inc, 1988.

36. Norton, John E. "Multiple Model Adaptive Tracking of Airborne Targets," MS Thesis, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, December 1988.

37. Parnas, David Lorge. "Software Aspects of Strategic Defense Systems," *Communications of the ACM, Vol 28 No 12*: 1326-1335 (December 1985).

38. Paulk, Mark C. "Problems with Distributed Ada Programs," *Proceedings of the 5th Annual International Phoenix Conference on Computers and Communications.* 396-400. New York: IEEE Press, 1986.

39. Press, William H., et al. *Numerical Recipes: the art of scientific computing* New York: Cambridge University Press, 1986.

40. Quinn, Micheal J. *Designing Efficient Algorithms for Parallel Computers* New York: McGraw-Hill, Inc, 1987.

41. Ramamoorthy, V. "Our Job is to Reduce the Errors," *Computer, Vol 19 No 11*: 64-65 (December 1986).

42. Rattner, Justin and William W. Lattin. "Ada Determines Architecture of 32-bit Microprocessor," *Electronics*: 119-126 (February 24, 1981).

43. Reynolds, P. F. and John S. Knight, et al. *The Implementation and Use of Ada on Distributed Systems with High Reliability Requirements*, March 1983. NASA Grant No. NAG-1-260.

44. Roark, Chuck and Ron Strauser, et al. "Supporting Ada Applications in a Distributed, Real-Time Environment," *Proceedings of the 1987 IEEE National Aerospace and Electronics Conference Volume 3.* 831-838. New York: IEEE Press, 1987.

45. Roubine, Oliver. "Reusable Software," *Application to Ada Higher Order Language to Guidance and Control.* NATO Advisory Group fro Aerospace Research and Development, 1986.

46. SofTech, Inc. *Designing Real-Time Systems in Ada* Final Report, 1986 (AD-A169687).

47. Stone, Harold S. *High-Performance Computer Architecture.* Addison-Wesley Publishing Company, 1987.

48. Tannenbaum, Andrew S. *Computer Networks* Englewood Cliffs: Prentice Hall, 1981.

49. Tannenbaum, Andrew S. and Robbert Van Renesse. "Distributed Operating Systems," *Computing Surveys, 17*: 419-470 (December 1985).

50. Tobin, David M. "A Multiple Model Adaptive Tracking Algorithm for a High Energy Laser Weapon System," MS Thesis, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, December, 1986.

51. Tobin, David M. and Peter S. Maybeck. "Substantial Enhancements to a Multiple Model Adaptive Tracking Algorithm for a High Energy Laser Weapon System," *Proceedings of the IEEE Conference on Decision and Control.* 2002-2011. New York: IEEE Press, 1987.

52. Waldrop, M. Mitchell. "Resolving the Star Wars Software Dilemma," *Science, 232*: 710-713 (9 May 1986).

53. Westermeier, T. F. "Parallel Processing Applied to Digital Flight Control Systems; Some Perspectives," *Proceedings of the IEEE 1981 National Aerospace and Electronics Conference.* 1010-1017. New York: IEEE Press, 1981.

54. Westermeier, T. F. and H. E. Hansen. "The use of Ada in Digital Flight Control Systems," *Proceedings of the AIAA Guidance and Control Conference.* 597-603. New York: AIAA Press, 1985.

55. Williams, Elizabeth. "Assigning Processes to Processors in Distributed Systems," *Proceedings of the 1983 IEEE International Conference on Parallel Processing.* 404-406. Silver Spring: IEEE Computing Society Press, 1983.

56. Yourdon, Edward. *Managing the Structured Techniques* New York: Yourdon, Inc, 1979.

Vita

Captain Walter J. Lemanski [redacted] May, 1980 a[redacted] received an appointment to the United States Air Force Academy where he studied computer science. Upon graduation in 1984, Captain Lemanski was awarded a Bachelor of Science degree, with honors, and a regular commission in the United States Air Force. He was assigned to Headquarters Military Airlift Command, serving from July 1984 to May 1986 as a systems analyst in the Software Applications Management Office and from June 1986 to May 1987 as the Executive Officer to the Director of Automation Support. Captain Lemanski entered the Air Force Institute of Technology in June 1987 to pursue the degree of Master of Science in Computer Engineering.

[redacted]

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release;distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S): AFIT/GCS/ENG/89M-2	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: School of Engineering; 6b. OFFICE SYMBOL (If applicable): AFIT/ENG	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State, and ZIP Code): Air Force Institute of Technology (AU) Wright-Patterson AFB, OH 45433-6583	7b. ADDRESS (City, State, and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: Strategic Defense Initiative Org; 8b. OFFICE SYMBOL (If applicable): S/PI	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code): The Pentagon Washington D.C. 20301-7100	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
PARALLEL ADA IMPLEMENTATION OF A MULTIPLE MODEL ADAPTIVE KALMAN FILTER TRACKING SYSTEM A SOFTWARE ENGINEERING APPROACH

12. PERSONAL AUTHOR(S)
Walter J. Lemanski, CAPT, USAF

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
MS Thesis	FROM _____ TO _____	March 1989	119

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	PARALLEL PROCESSING KALMAN FILTERING SOFTWARE ENGINEERING ADA PROGRAMMING LANGUAGE
12	05		
20	03		

19. ABSTRACT (Continue on reverse if necessary and identify by block number)
Dr Thomas Hartrum

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT: ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION: UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL: Dr Thomas Hartrum	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL: AFIT/ENG

DD Form 1473, JUN 86 — Previous editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

The success of the Strategic Defense Initiative depends directly on significant advances in both computer hardware and software development technologies. Parallel architectures and the Ada programming language have advantages that make them candidates for use in SDI command and control computer systems. This thesis examines those advantages in the context of an SDI-type application: the implementation of a Kalman filter tracking system.

This research consists of three parts. The first is a set of software engineering guidelines developed for use in creating parallel designs suitable for implementation in Ada. These guidelines cover the design process from initial problem analysis to final detailed design. Methods of problem decomposition are discussed, as are language partitioning strategies. Justification is provided for using the Ada task construct for process boundaries, and Ada multitasking design issues are reviewed. A parallel software design methodology is also described.

The second part is an illustration of the use of these software engineering guidelines to design a multiple model adaptive Kalman filter based tracking system and its associated simulation program. Theoretical research, currently being conducted at the Air Force Institute of Technology, is used as a basis for this phase. The result of each step of the guidelines is documented.

The final part is a parallel implementation of the Kalman filter tracking system design, including several reusable matrix operation routines, on the Encore Multimax 320 in Ada. Implementation and testing techniques are documented, as are problems that were encountered with the design and the Encore Concurrent Ada development environment. Timing results are analyzed to determine the efficiency of the design and implementation, as well as the suitability of the particular tracking algorithm for parallelization. Additional analysis is included on the adequacy of the software engineering guidelines developed in part one and on the results of using Ada in a true parallel environment.